Títol: Proposta de gestió del sistema operatiu per a arquitectures heterogènies multiprocessador

Volum: 1 de 1

Alumne: Javier Joglar Huber

Director/Ponent: Marisa Gil Gómez

Departament: Arquitectura de Computadors

Data: 30 de juny de 2008

DADES DEL PROJECTE

Títol del Projecte: Proposta de gestió del sistema operatiu per a arquitectures heterogènies multiprocessador
Nom de l'estudiant: Javier Joglar Huber
Titulació: Enginyeria en Informàtica
Crèdits: 37.5
Director/Ponent: Marisa Gil Gómez
Departament: Arquitectura de Computadors (AC)

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: Nacho Navarro Mas

Vocal: Lluís Vila Grabulosa

Secretari: Marisa Gil Gómez

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data: 30 de juny de 2008

ÍNDEX

ÍNDEX DE FIGURES

ÍNDEX DE TAULES

1. INTRODUCCIÓ

En un mercat que evoluciona tant ràpidament com és el de l'arquitectura de computadors, la importància de disposar d'un sistema operatiu que sigui capaç d'aprofitar la potència que ofereix el *hardware* és cada cop més important. La integració de diversos processadors dins d'un mateix xip, ja siguin processadors idèntics o no, fa que les característiques del sistema operatiu s'hagin d'adaptar convenientment, per treure'n el màxim profit. Per exemple, els actuals Intel® Core2Duo [1] i Core2Quad [2] disposen de 2 i 4 nuclis, respectivament. Un altre exemple seria combinar dins un mateix xip CPU's i GPU's [3], ja que ara per ara les unitats GPU són la millor opció per a executar operacions en coma flotant.

Fins fa relativament poc, els computadors es dissenyaven seguint una arquitectura Von Neumann, és a dir, una unitat de control, un bloc de memòria, i una unitat aritmètica que rebia unes entrades i generava unes sortides, tot això comunicat per un bus d'interconnexió que feia visible tots els elements entre ells. Actualment, gràcies a l'avanç de la tecnologia CMOS (*Complementary metal-oxide-semiconductor*) i el disseny VLSI (*Very-large-scale integration*), es poden dissenyar sistemes més petits, amb menys consum i amb més potència. Per això, enlloc d'haver-hi un sol processador en podem disposar de més d'un, i cada processador pot tenir la seva memòria local. Totes aquestes modificacions a nivell de *hardware* impliquen modificacions a les aplicacions. Per això, apareixen nous models de programació, llibreries i propostes a nivell de sistema operatiu per aconseguir-ho.

De la mateixa manera, cada cop més freqüentment aquests processadors no són idèntics entre sí, sinó que cadascun d'ells està especialitzat en determinades tasques[1], fet que proporciona un rendiment encara major en determinats entorns. Acostumen a ser una combinació de processadors, uns més generals i altres més específics, fent que estiguin especialitzats en algun tipus de tasca. Aquests processadors s'anomenen heterogenis, a diferència dels presentats anteriorment, anomenats homogenis. Aquesta sembla que serà la tendència en els propers anys, i és per això que s'han de proposar alternatives per a aprofitar al màxim aquestes tecnologies.

[1] Un exemple d'això és el processador Cell BE, desenvolupat per IBM, que consta d'un processador de propòsit general i vuit processadors específics. Aquest processador és el nucli de la consola PlayStation 3.

En aquest sentit, disposarem de processadors especialitzats en unes determinades tasques (càlculs de nombres en coma flotant com ara les GPU's, operacions d'entrada/sortida, processadors amb instruccions SIMD[2], etc.). Una aplicació pot beneficiar-se d'aquest fet si una part del seu codi (ja sigui un bucle, una funció o un conjunt de funcions) presenta alguna característica algorísmica d'aquest estil. És feina del sistema operatiu proporcionar un mecanisme per a que el programador pugui indicar aquestes característiques, i del propi sistema operatiu aprofitar-les per a obtenir un major rendiment a l'hora d'executar aquestes aplicacions, planificant totes les tasques i decidint en temps d'execució on és més adient executar una aplicació, tenint en compte les seves característiques i la càrrega de la màquina. Per exemple, si el programador vol fer una multiplicació de matrius de nombres en coma flotant, i la màquina disposa d'una unitat GPU a on es pot enviar aquesta tasca, s'obtindrà més rendiment que si aquesta multiplicació es fa a un processador de propòsit general. Per tant, el sistema podria decidir en temps d'execució qui és l'encarregat de fer aquest càlcul coneixent l'estat del recursos i la idoneïtat que l'execució es dugui a terme en un nucli o en un altre.

Com es pot observar, el concepte de CPU (*Central Processing Unit*) està canviant, donat que el terme es referia a una única unitat de procés en una arquitectura von Neumann, a diferència del que està apareixent actualment al mercat. Per tant, el terme més adequat per referir-nos actualment als processadors actuals seria quelcom més genèric com ara "elements de procés".

El camp dels multiprocessadors heterogenis (diferents tipus de processador dins el mateix xip) és un dels camps de recerca més actius actualment en l'arquitectura de computadors, donat que és una tecnologia molt nova i que encara necessita eines i propostes per a aprofitar-la. En aquest sentit, sembla que el *hardware* vagi un pas per davant del *software*, i facin falta propostes per aprofitar aquestes novetats del mercat. Alguna de les propostes ja existents, tant a nivell d'aplicació com de sistema operatiu, són:

- CESOF (*CBE Embedded SPE Object Format*) [5]

[2] *Single-Instruction Multiple-Data*, quan s'ha d'aplicar la mateixa operació per a un conjunt molt gran de dades. Cada tipus de processador té el seu propi repertori (3DNow! a AMD, SSE a Intel).

- EXOCHI (*Exoskeleton Sequencer C for Heterogeneous Integration*) [6]
- CellSs (*Cell Superscalar*) [7]

Però com ja s'ha dit anteriorment, el *hardware* evoluciona d'una manera més ràpida que el *software*, donat que no és senzill programar sense saber exactament on s'executarà el programa, i tot i sabent-ho no és fàcil programar amb paral·lelització real, és a dir, distribuir l'execució del programa entre totes les unitats d'execució disponibles. A més, el fet de ser multiprocessadors heterogenis és una dificultat afegida. Així doncs, aquest sembla que és i serà el gran repte de la computació dels propers anys.

Sent conscients d'aquesta situació, en aquest Projecte es proposa una adaptació del sistema operatiu per a aprofitar un escenari com el presentat anteriorment. El primer pas serà desenvolupar el Projecte en una arquitectura multiprocessador homogènia (pel fet de ser més coneguda), i posteriorment s'adaptarà per a una arquitectura multiprocessador heterogènia (Cell BE) per veure que la proposta pot aprofitar-se en un entorn heterogeni.

Fent-ho així, serem conscients també de què caldria fer per si, en un futur, volguéssim afegir noves arquitectures al sistema. Per això, totes les decisions que es prenguin a mesura que avança el Projecte s'han de veure també des de l'òptica d'una arquitectura com la del Cell BE, i inclús pensant que en un futur no molt llunyà podrien dissenyar-se xips amb distribucions de processadors i acceleradors molt complexes. Per tant, a l'avaluar diferents alternatives, cal triar la que també a posteriori sigui més adequada i ens faciliti més la feina.

Alhora, aquest Projecte es desenvolupa paral·lelament, i de forma complementària, a un altre Projecte Final de Carrera. La proposta és comuna als dos Projectes (donat que aquesta idea s'ha treballat conjuntament amb la directora d'ambdós Projectes durant molts mesos abans de la realització d'aquests), però a l'hora d'implementar-lo s'han dividit les tasques perquè cadascú sigui el responsable d'una part. En concret, l'altre projecte es centra més en la part d'usuari, a diferència d'aquest que es centra en la part del sistema operatiu. Fent-ho així, el disseny es realitzarà completament i es podrà avaluar la proposta globalment.

Per tant, quan s'expliquin les fases de desenvolupament del Projecte (capítols 5, 6 i 7) s'analitzaran tots els aspectes implicats (per poder entendre la proposta globalment) però no s'aprofundirà tant en aspectes de nivell d'usuari com en els de sistema operatiu, perquè aquesta diferència de responsabilitats quedi clara.

A continuació es presenten els objectius marcats abans de començar el Projecte. Posteriorment, les motivacions a nivell personal per a desenvolupar un Projecte com aquest, l'estructura del document per a la seva fàcil comprensió, i, finalment, uns agraïments.

1.1. Objectius

Com ja s'ha comentat, l'objectiu principal del Projecte és aprofitar la potència que, cada cop més, el *hardware* ofereix a l'hora d'executar aplicacions, donat que la tendència actual del mercat és integrar, dins un mateix xip, diversos processadors. Per fer això, cal adaptar convenientment el *software*, a diferents nivells, per a reflectir aquestes noves situacions.

Per una banda, i donat que el que volem optimitzar són les aplicacions que s'executen, aquestes s'han de modificar convenientment, permetent que el programador (també podria ser el propi compilador qui ho detectés, però ens desviaríem del nostre propòsit) indiqui quina o quines parts del codi es poden executar en una arquitectura específica per a treure'n un major rendiment. Per fer això, cal conèixer bé el format executable utilitzat per a poder decidir on i com guardar aquesta informació a l'executable. També cal conèixer amb detall i dominar els passos de compilació i enllaç d'un programa, involucrats en el procés de creació de l'executable final.

Per altra banda, seria convenient oferir una capa intermèdia, que sigui el nexe entre la part d'aplicació i el sistema operatiu. Aquesta funció la realitzarà una llibreria, que farà que la comunicació entre les aplicacions i el sistema operatiu sigui més flexible. És a dir, ens ajudarà a fer que la proposta sigui més fàcil d'analitzar, detectar possibles errors i afegir noves funcionalitats. Serà el nexe d'unió, també, entre els dos Projectes Finals de Carrera, ja que és la via de comunicació entre la capa d'aplicació i el sistema operatiu. Un darrer objectiu referent a aquesta llibreria és que la seva interfície hauria de ser similar a alguna

proposta existent, per facilitar-ne l'ús als programadors.

Si es vol millorar algun aspecte més lligat a l'arquitectura, com és el nostre cas, el punt clau és el sistema operatiu, donat que és qui té la visió més amplia dels recursos que disposa la màquina i del profit que en pot treure. Per tant, és necessari adaptar el sistema operatiu per a que reconegui i interpreti les modificacions introduïdes al binari, disposi d'uns nous objectes que representin aquesta informació associada a un programa en execució, i dugui a terme una gestió de recursos més eficient i eficaç. Això implica, també, tenir un coneixement avançat de les estructures de dades que representen i maneguen els processos i threads al sistema operatiu, els passos que es segueixen per a carregar un procés a memòria, la interacció entre el nivell d'usuari i el nucli, la planificació de tasques que duu a terme per anar canviant l'execució entre els diferents processos, etc.

De la mateixa manera, ha de proveir a la llibreria de tota la informació que aquesta necessiti per a desenvolupar les seves tasques correctament, mitjançant noves crides a sistema.

Més concretament, el que es pretén es proporcionar a les aplicacions una interfície amb el sistema operatiu, de tal manera que el programador indiqui que vol executar un determinat codi, i amb la informació que prèviament ha afegit indicant les característiques d'aquest codi, el sistema operatiu sigui capaç d'interpretar-ho i enviar aquest codi al processador o accelerador més adient per a executar-ho. D'aquesta manera, el thread podrà anar "saltant" entre els diferents processadors, canviant el seu context d'execució.

Això implica, per una banda, oferir tots els mecanismes de comunicació amb les aplicacions per a poder indicar la funció a executar, els seus paràmetres d'entrada i de sortida, i per altra banda un coneixement per part del sistema operatiu de l'estat dels recursos de la màquina, per a planificar correctament aquesta execució.

Ha de quedar clar que no estem definint un nou model d'execució. El que estem fent és oferir una abstracció perquè el programador pugui enviar un codi a executar-se a un processador o accelerador sense que s'hagi de preocupar d'altres aspectes. Però si les característiques d'aquest processador impliquen que es creï, per exemple, un thread

addicional que sigui l'encarregat de gestionar aquesta execució, nosaltres també ho farem. El que sí farem és estudiar models d'execució i modificar lleugerament el seu comportament.

Per a poder dur a terme les modificacions que volem, és clar que necessitem disposar del codi font del sistema operatiu. Per això, la implementació es durà a terme en un entorn Linux. Concretament, modificarem la versió actual del nucli (2.6.24) donat que és la que dóna més suport al nou *hardware*. La distribució que s'ha escollit ha estat Fedora Core [4], perquè disposa de versions tant per màquines x86 com per PowerPC, arquitectures on es desenvoluparà el Projecte.

Una altra cosa que cal tenir present és que la manera de representar un procés a dins del sistema operatiu ha anat canviant al llarg del temps. En un principi, els objectes representaven un procés, i a mesura que el *hardware* ha evolucionat, s'ha incorporat el concepte de thread (un o més threads pertanyen al mateix procés, amb totes les implicacions de compartició de dades i paral·lelisme que això comporta) per a aprofitar al màxim les possibilitats que ofereix el *hardware* d'executar alhora diferents threads. Per això, cal conèixer amb detall aquests nous objectes del sistema operatiu per aprofitar-los convenientment.

Una vegada s'ha donat la primera visió de la proposta, cal dir que un dels objectius principals és dotar al sistema d'una gran flexibilitat, de tal manera que sigui relativament senzill afegir una nova arquitectura al sistema que pogués sorgir, seguint el mateix esquema que es proposa. Per tant, no es persegueix tant l'obtenció d'un gran rendiment a l'hora d'executar aplicacions com proporcionar una interfície clara, còmoda i flexible per a fer que les aplicacions puguin explotar al màxim les capacitats del *hardware*. Volem que el programador pugui indicar d'una manera molt clara que un tros de codi s'executi en un nucli en concret, sense que s'hagi de preocupar d'altres aspectes que són intrínsecs a aquesta situació (carregar el codi al processador, les dades, etc.).

Finalment, s'hauran d'estudiar les diferents alternatives de disseny, avaluar el seu rendiment, mesurar la sobrecàrrega afegida i el guany obtingut en l'execució de les aplicacions.

Com es pot veure, hi ha moltes disciplines implicades en aquesta proposta, i a mesura que es vagin introduint temes encara sorgiran més aspectes a tenir en compte. És per això que, tot i que la proposta és complerta i s'han pensat les seves implicacions, només s'ha implementat una petita part de tot això. El temps limitat que representa un Projecte Final de Carrera no és suficient per estudiar tots els camps que s'han presentat i implementar tot el que s'ha explicat. Però ha de quedar clar que (com indica el propi nom del Projecte) es presenta una proposta, no un producte definitiu totalment funcional. S'implementarà una petita part, però on més s'ha incidit és en fer una proposta.

1.2. Motivacions personals

Personalment, les motivacions que tinc per dur a terme aquest Projecte Final de Carrera són diverses i variades.

En primer lloc, el camp de l'arquitectura de computadors i els sistemes operatius és el que més m'agrada i m'interessa de tots els que he estudiat al llarg de la carrera. És per això que el Projecte tracta sobre aquest tema.

En segon lloc, el camp dels multiprocessadors heterogenis és un camp molt actual, i sembla que amb molt futur. Per això volia aprofundir més els coneixements adquirits a la carrera sobre multiprocessadors, i aprendre sobre arquitectures heterogènies, desconegudes per mi fins a la realització d'aquest Projecte. També m'interessava poder treballar amb el processador Cell, del qual ja n'havia sentit a parlar molt al llarg de la carrera.

Finalment, volia introduir-me una mica més en el camp de la recerca, ja que estic duent a terme un Programa de Formació d'Estudiants en Departaments i Instituts[3] des del mes de setembre de l'any 2007, que porta per títol *Operating System and Runtime Management Proposals for Heterogeneous Multicore Systems*. Anteriorment, també havia

[3] "L'objectiu d'aquest Programa és oferir una sòlida formació de l'estudiant en els mètodes i procediments propis de la recerca, el desenvolupament i el treball professional, més que l'obtenció de resultats immediats lligats a les activitats dels departaments o instituts.". Extret de: http://www.ice.upc.edu/documents/cg_formacio_centres.pdf

col·laborat amb el Departament d'Arquitectura de Computadors, aproximadament des del febrer de l'any 2007. Per això, un Projecte Final de Carrera enfocat a la recerca m'atreia més.

1.3. Estructura de la memòria

La memòria s'ha estructurat de la següent manera:

1. En aquest primer capítol, es fa una introducció del tema del treball, el motiu de l'estudi d'aquesta disciplina de l'arquitectura de computadors, els objectius que es volen assolir, les motivacions personals per dur-lo a terme i uns agraïments personals.

2. Al segon capítol es presenta la planificació temporal i un anàlisi dels costos econòmics.

3. Al tercer capítol s'analitzen els antecedents existents avui dia, observant les propostes per a multiprocessadors heterogenis tant a nivell d'aplicació com de sistema operatiu, i la seva factibilitat.

4. Al quart capítol s'expliquen uns conceptes generals que ajudaran a entendre amb més profunditat la resta de capítols, incidint sobretot en la part referent al sistema operatiu.

5. Al cinquè capítol es realitza una primera aproximació al Projecte, començant a presentar les propostes realitzades i detalls d'implementació.

6. Al sisè capítol es continuarà amb l'explicació del Projecte, partint de les propostes presentades al capítol anterior i millorant els aspectes que ho necessiten, per concloure amb la implementació de tot el sistema en una arquitectura multiprocessador homogènia x86. Aquest serà el primer pas per la completa realització del Projecte Final de Carrera.

7. Al setè capítol es migrarà la proposta a una arquitectura multiprocessador heterogènia com és el Cell BE, presentant les modificacions necessàries per a fer-ho en dues iteracions, i detalls específics que han sorgit a mesura que s'ha adaptat la proposta.

8. Al vuitè capítol es presenten les conclusions a les que s'ha arribat realitzant aquest Projecte, comparant-les amb els objectius marcats a l'inici del Projecte.

9. Al novè capítol s'introdueix el treball futur indicant què s'hauria o es podria realitzar, amb el camí a seguir i les idees que es podrien desenvolupar, si es volgués continuar amb la implementació del sistema.

10. Al desè i darrer capítol es presenten les referències que s'han introduït a la memòria, que han sigut gran part de la bibliografia consultada per dur-la a terme.

1.4. Agraïments

En primer lloc vull agrair als meus pares tot el suport que, al llarg d'aquest Projecte i de tota la carrera, m'han brindat.

Aquest Projecte s'ha desenvolupat amb l'ajuda de Marisa Gil i Nacho Navarro, professors del Departament d'Arquitectura de Computadors de la Universitat Politècnica de Catalunya. A ells he d'agrair, en segon lloc, l'haver desenvolupat aquest Projecte, especialment a la Marisa Gil per ser també la directora del Programa de Formació que he dut a terme.

En tercer lloc, agrair a Judit Planas la seva ajuda en parts del meu Projecte, tot i que ella ha desenvolupat el complementari a aquest, més corresponent a la part d'aplicació.

Finalment, agrair també l'ajuda de Lluc Álvarez i Julio Merino amb tota la part corresponent al Cell BE. Sense ells, aquesta part hagués estat molt més costosa del que finalment ha estat.

2. PLANIFICACIÓ I ANÀLISI ECONÒMIC

En aquest capítol es presenta la planificació del Projecte, juntament amb un anàlisi econòmic. La planificació està feta tenint en compte que només ha desenvolupat el Projecte una persona. Com ja s'ha comentat a la introducció, hi ha un Projecte complementari a aquest, i també caldria tenir en compte aquest fet.

Per altra banda, també es podria fer una planificació si el Projecte l'hagués de desenvolupar un equip de treball, seguint les fases de desenvolupament de l'enginyeria del software (Anàlisi, Iteracions, Test, etc.), i amb uns rols determinats (Cap de Projecte, Analista, Programador, etc.). No s'ha fet així perquè, per la naturalesa del Projecte, no és necessari aquesta organització.

Al ser un Projecte de recerca, és difícil preveure les etapes i la duració d'aquestes, ja que depenen de com vagi evolucionant l'estudi que es fa. El mateix estudi fa que una etapa que en principi podia semblar més costosa no ho sigui tant, o a l'inrevés. Tot i això, s'ha intentat establir un ordre de feines a realitzar, amb la seva durada aproximada. De la mateixa manera, els objectius s'han anat adaptant a mesura que ha avançat el Projecte, fent que la planificació s'hagués d'anar adaptant.

La planificació del Projecte s'ha dividit en tres etapes, que representen les tres iteracions amb les que ens anirem aproximant progressivament a la resolució dels objectius. Aquestes tres iteracions corresponen als capítols 5, 6 i 7 d'aquesta memòria, i representen la major part del treball realitzat durant aquests mesos. Per tant, es dóna per fet que ja s'ha pensat la proposta (tal i com va ser, ja que la proposta és anterior al desenvolupament del que ha estat pròpiament el Projecte) i només es mostren les etapes d'implementació.

En aquest capítol no s'expliquen amb detall les diferents tasques de cada etapa, donat que caldria anticipar conceptes que s'introduiran a mesura que s'avanci en l'explicació de la proposta. Si es vol entendre amb profunditat les etapes que s'han seguit, caldria tornar a veure els diagrames de temps una vegada s'ha conclòs la lectura d'aquest document. A cadascun dels tres diagrames es mostra l'etapa desglossada corresponent, juntament amb el nom de les altres dues etapes.

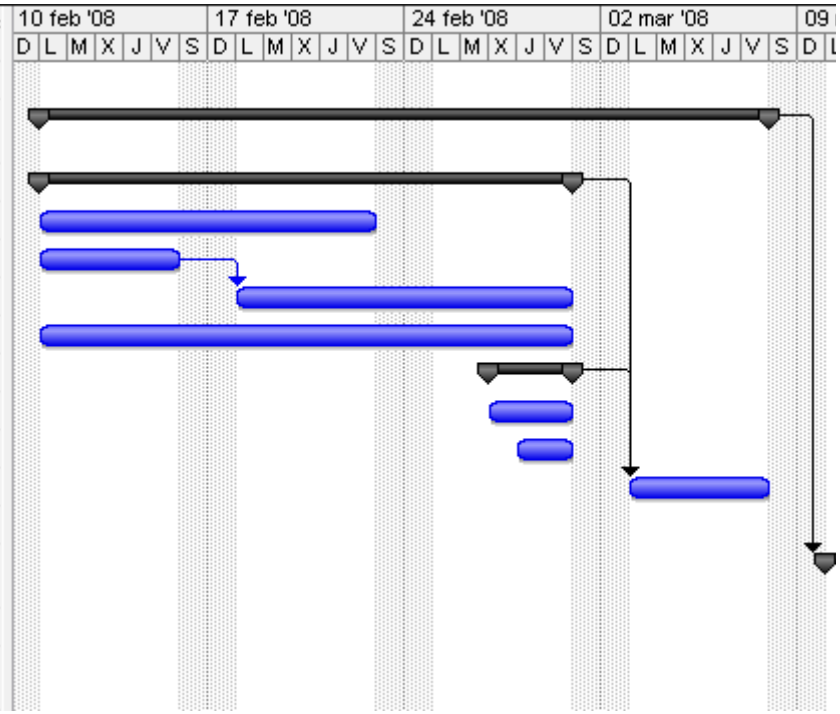
El Projecte Final de Carrera té una càrrega de 37.5 crèdits. Segons la guia docent de la FIB[4], equival a un quadrimestre a temps complert (40 hores a la setmana durant 15 setmanes). Això suposen 600 hores de treball. Per tant, suposant que el cost d'una hora de treball per persona és de 30€, el cost econòmic del Projecte seria de 600h x 30€/h = 18.000€.

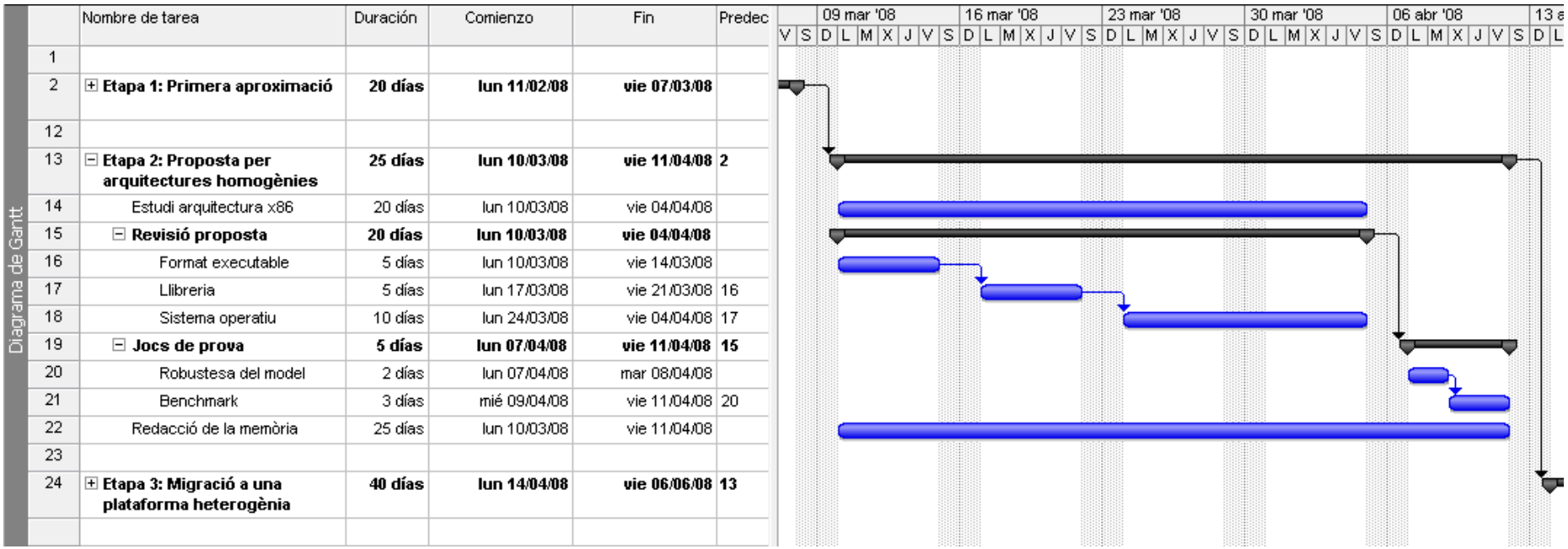
Si també tenim en compte el cost del *hardware* a on s'ha desenvolupat el Projecte (el cost aproximat d'una PlayStation 3, que conté el processador Cell, és d'uns 400€, i el cost d'un PC com l'utilitzat és d'uns 600€), dóna un cost total de 19.000€.

[4] http://www.fib.upc.edu/fib/infoAca/estudis/PFC/normativa/2.html

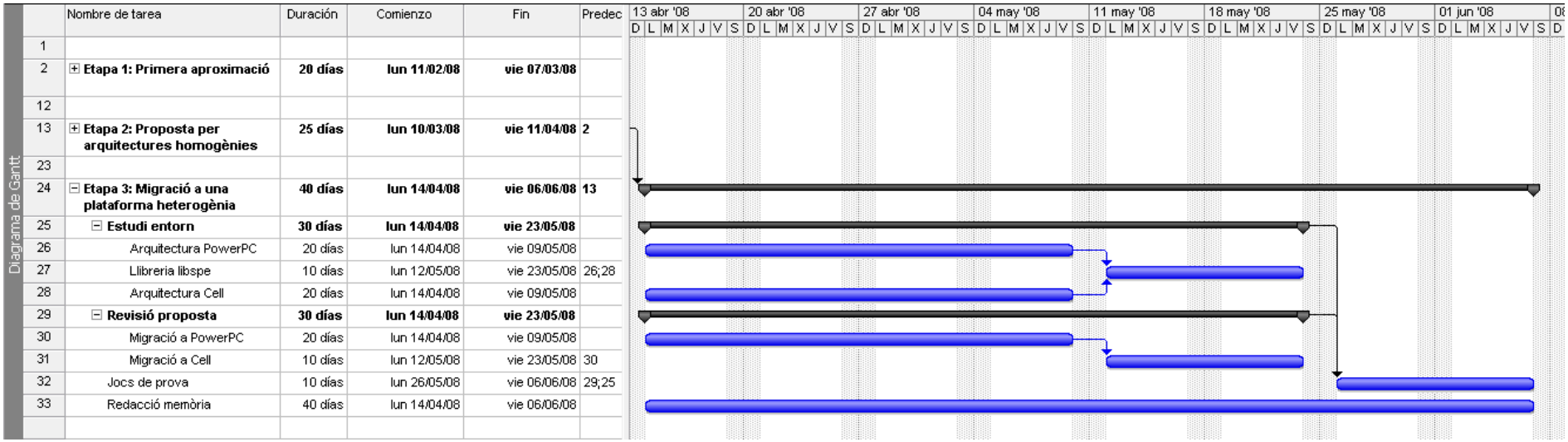
Diagrama de Gantt

	Nombre de tarea	Duración	Comienzo	Fin	Predec
1					
2	**Etapa 1: Primera aproximació**	**20 días**	**lun 11/02/08**	**vie 07/03/08**	
3	**Estudi entorn i proposta**	**15 días**	**lun 11/02/08**	**vie 29/02/08**	
4	Formats executables	10 días	lun 11/02/08	vie 22/02/08	
5	Llibreries dinàmiques	5 días	lun 11/02/08	vie 15/02/08	
6	Llibreria pthread	10 días	lun 18/02/08	vie 29/02/08	5
7	Lectura bibliogràfica	15 días	lun 11/02/08	vie 29/02/08	
8	**Redacció**	**3 días**	**mié 27/02/08**	**vie 29/02/08**	
9	Informe del Projecte	3 días	mié 27/02/08	vie 29/02/08	
10	Índex del Projecte	2 días	jue 28/02/08	vie 29/02/08	
11	Avaluació i prova	5 días	lun 03/03/08	vie 07/03/08	3;8
12					
13	**Etapa 2: Proposta per arquitectures homogènies**	**25 días**	**lun 10/03/08**	**vie 11/04/08**	**2**
23					
24	**Etapa 3: Migració a una plataforma heterogènia**	**40 días**	**lun 14/04/08**	**vie 06/06/08**	**13**





	Nombre de tarea	Duración	Comienzo	Fin	Predec
1					
2	**Etapa 1: Primera aproximació**	**20 días**	**lun 11/02/08**	**vie 07/03/08**	
12					
13	**Etapa 2: Proposta per arquitectures homogènies**	**25 días**	**lun 10/03/08**	**vie 11/04/08**	2
14	Estudi arquitectura x86	20 días	lun 10/03/08	vie 04/04/08	
15	**Revisió proposta**	**20 días**	**lun 10/03/08**	**vie 04/04/08**	
16	Format executable	5 días	lun 10/03/08	vie 14/03/08	
17	Llibreria	5 días	lun 17/03/08	vie 21/03/08	16
18	Sistema operatiu	10 días	lun 24/03/08	vie 04/04/08	17
19	**Jocs de prova**	**5 días**	**lun 07/04/08**	**vie 11/04/08**	15
20	Robustesa del model	2 días	lun 07/04/08	mar 08/04/08	
21	Benchmark	3 días	mié 09/04/08	vie 11/04/08	20
22	Redacció de la memòria	25 días	lun 10/03/08	vie 11/04/08	
23					
24	**Etapa 3: Migració a una plataforma heterogènia**	**40 días**	**lun 14/04/08**	**vie 06/06/08**	13



	Nombre de tarea	Duración	Comienzo	Fin	Predec
1					
2	**Etapa 1: Primera aproximació**	**20 días**	**lun 11/02/08**	**vie 07/03/08**	
12					
13	**Etapa 2: Proposta per arquitectures homogènies**	**25 días**	**lun 10/03/08**	**vie 11/04/08**	2
23					
24	**Etapa 3: Migració a una plataforma heterogènia**	**40 días**	**lun 14/04/08**	**vie 06/06/08**	13
25	**Estudi entorn**	**30 días**	**lun 14/04/08**	**vie 23/05/08**	
26	Arquitectura PowerPC	20 días	lun 14/04/08	vie 09/05/08	
27	Llibreria libspe	10 días	lun 12/05/08	vie 23/05/08	26;28
28	Arquitectura Cell	20 días	lun 14/04/08	vie 09/05/08	
29	**Revisió proposta**	**30 días**	**lun 14/04/08**	**vie 23/05/08**	
30	Migració a PowerPC	20 días	lun 14/04/08	vie 09/05/08	
31	Migració a Cell	10 días	lun 12/05/08	vie 23/05/08	30
32	Jocs de prova	10 días	lun 26/05/08	vie 06/06/08	29;25
33	Redacció memòria	40 días	lun 14/04/08	vie 06/06/08	

3. ANÀLISI D'ANTECEDENTS I FACTIBILITAT

En aquest capítol s'analitzen els antecedents i les propostes existents a dia d'avui, al camp dels multiprocessadors heterogenis, tant a nivell d'aplicació com de sistema operatiu. Veurem quines alternatives existeixen per a explotar al màxim les capacitats d'aquests sistemes en aquests dos nivells.

També analitzarem la factibilitat d'aquest Projecte, tenint en compte que al ser un tema molt actual encara s'estan investigant diferents solucions per adreçar el problema.

3.1. Anàlisi d'antecedents

Actualment no existeixen gaires precedents a l'hora d'aprofitar l'heterogeneïtat dels processadors dins d'un mateix xip. En canvi, sí que existeixen moltes propostes per a aprofitar el fet de disposar de multiprocessadors. Això és degut a que el mercat de processadors heterogenis és molt més recent que el dels multiprocessadors. Mostrarem com està actualment el mercat en aquests dos aspectes, observant-los des del punt de vista de les propostes fetes a la part de les aplicacions i al sistema operatiu.

3.1.1. Propostes per les aplicacions

Quant a les aplicacions, existeixen moltes propostes perquè el programador indiqui que un tros de codi es pot executar en paral·lel per aprofitar un escenari multiprocessador. Exemples d'aquestes propostes podrien ser:

- *Pthreads*: Una llibreria que defineix una API per crear i manipular threads. Una aplicació pot disposar d'un o més threads, que no són més que fils d'execució. A nivell de programador, un thread és una funció que es pot executar independentment de la resta del programa. Cada thread disposa de la seva pila, els seus registres i propietats per la planificació (política, prioritat, etc.). Amb aquesta informació és possible planificar un thread paral·lelament al seu "pare" en un altre processador. Tota la informació i els recursos són compartits entre tots els threads de la mateixa aplicació, fent que sigui poc costós disposar de molts threads, ja que gran part de la feina no s'haurà de fer de nou pel fet de ser informació compartida. D'aquí que també s'anomenin *Lightweight Processes*. A l'apartat 5.3.2 s'entra més

en detall sobre el funcionament d'aquesta llibreria.

- *OpenMP*: Open Multi-Processing [8]. És una API que defineix un model de programació que ens permet afegir concurrència a les aplicacions en sistemes de memòria compartida.
- *MPI*: Message Passing Interface [9]. És una tècnica emprada en sistemes on la memòria no és compartida, i intervé el pas de missatges entre els diferents threads que s'executen en paral·lel.
- *PVM*: Parallel Virtual Machine. És una llibreria que fa que una xarxa de computadors comparteixi els seus recursos per aprofitar-los en l'execució d'una aplicació.
- *Servo*: *Service Oriented programming model* [10]. És un model de programació per solucionar el problema de l'escalabilitat en sistemes multiprocessador, on es descompon el programa en diferents components, que a la seva vegada són encapsulats en serveis. Un servei representa una tasca a executar pel *hardware*.

En canvi, no existeixen gaires propostes per a aprofitar la heterogeneïtat dels processadors d'un xip. Una de les propostes existents és un format binari, que ja s'ha introduït anteriorment, i s'anomena CESOF (*CBE Embedded SPE Object Format*) [5]. Aquest format binari s'utilitza al Cell, i per això s'explica amb més detall tot seguit, perquè s'utilitzarà quan es migri el Projecte al Cell.

El processador Cell, desenvolupat per IBM, Sony i Toshiba, és un multiprocessador heterogeni, que disposa d'un nucli SMT[5] que gestiona dos fils d'execució (anomenat *Power Processing Element,* PPE) i vuit coprocessadors especialitzats en operacions vectorials i multimèdia a gran velocitat (anomenats *Synergistic Processing Element,* SPE). Els processadors estan connectats per un bus intern amb un gran ample de banda anomenat *Element Interconnect Bus* (EIB). Es pot trobar mes informació sobre el Cell a [11] i a l'apartat 7.2 d'aquesta memòria, on s'explica amb més detall. En aquest punt només volem donar els mínims detalls i nomenclatures per poder explicar el format CESOF.

La idea del format binari CESOF és incloure la part que ha d'executar el SPE

[5] *Simultaneous Multithreading*, tècnica que permet executar múltiples fils d'execució simultàniament.

incrustada a dins el mateix executable, de tal manera que el PPE sap que en aquella part de l'executable hi té el codi pels SPE's. La Figura 1 mostra com es duu a terme al Cell.

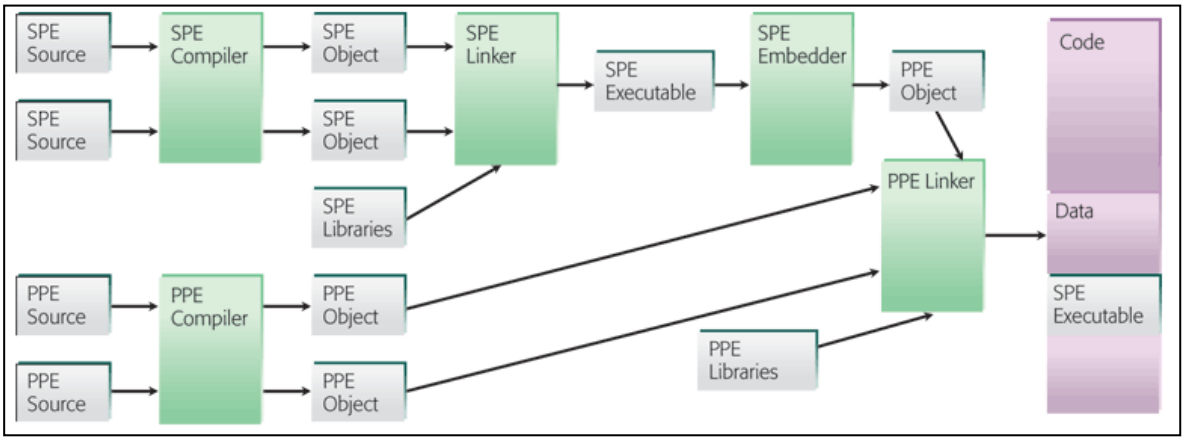


Figura 1: Obtenció de l'executable utilitzat al Cell (extreta de [12])

El codi font de la part que s'executarà a l'SPE es compila amb un compilador propi i els objectes generats, juntament amb les llibreries que utilitzin, s'enllacen formant l'executable SPE. Aquest executable es transforma en un fitxer objecte PPE.

Per altra banda, la part corresponent al PPE també es compila per generar fitxers objecte, que juntament amb les llibreries i el fitxer objecte que representa l'executable SPE s'enllacen formant l'executable final. D'aquesta manera, l'executable SPE queda incrustat dins d'una secció de dades del binari que s'executarà al PPE.

En aquest sentit, la part de l'executable que correspon a l'SPE és desconeguda pel PPE, ja que és una ISA diferent a la que ell utilitza (són un conjunt de 0's i 1's no interpretables), però només necessita conèixer en quina secció ho té, i quant ocupa, per a enviar-li a l'SPE quan aquest l'ha d'executar.

No només el CESOF utilitza aquesta tècnica, anomenada *fat binary*, d'incloure codi de diferents ISA's a dins del mateix executable. Per exemple, també existeixen propostes d'Apple [13] per utilitzar-la.

3.1.2. Propostes pel sistema operatiu

Quant al sistema operatiu, el fet de disposar de diversos processadors ja és aprofitat per la gran majoria de sistemes operatius, i l'únic que cal fer és anar distribuint l'execució dels diferents processos entre ells. No entrem aquí en temes com ara l'afinitat d'un procés a un processador (si un procés es va executar en un processador abans d'haver fet un canvi de context d'execució, quan s'ha de tornar a planificar és preferible que es torni a enviar al processador on va ésser executat per si es pot aprofitar informació localitzada a les caches del processador), que ens farien desviar del nostre propòsit. Si es volen conèixer més detalls, es poden consultar a [14].

En canvi, no existeixen tampoc moltes propostes per a aprofitar la heterogeneïtat dels processadors. Una d'elles ja s'ha comentat anteriorment, i s'anomena CellSs (Cell Superscalar) [7].

El CellSs, desenvolupat per membres de la UPC i del Barcelona Supercomputing Center, és un model de programació per aprofitar l'execució de tasques al Cell. La idea és que el programador anoti funcions que poden ser executades en paral·lel, i en temps d'execució es construeix un graf de dependències per a planificar l'execució en paral·lel de les diferents funcions. No obstant, el que ara ens interessa és el que es fa internament. Quan el sistema operatiu detecta que es vol executar una d'aquestes funcions, esperarà a que hi hagi un SPE lliure (si es que no n'hi ha cap) per a enviar-li la tasca o l'executarà directament al PPE.

Una altra proposta existent per gestionar la heterogeneïtat a nivell de sistema operatiu, en arquitectures multiprocessador heterogènies el va realitzar un alumne de la facultat en un Projecte Final de Carrera, que es pot trobar a [15]. Aquest Projecte està més enfocat a homogeneïtzar la visió que el sistema operatiu té dels processadors d'una màquina (en concret, del Cell BE), ja que actualment no es tracten igual els processadors de propòsit general que els coprocessadors específics. Aquests darrers es veuen com dispositius d'entrada/sortida, no com unitats d'execució. Per tant, aquest Projecte podria ser útil si volem fer una proposta de gestió d'aquests processadors, per tenir una visió més similar de tots ells entre sí.

3.2. Anàlisi de factibilitat

Com ja s'ha comentat anteriorment, aquest Projecte Final de Carrera s'engloba a dins de la realització d'un Programa de Formació d'Estudiants en Departaments i Instituts, enfocat en la introducció de l'estudiant en el camp de la recerca i la investigació.

La metodologia pròpia de la recerca és la cerca bibliogràfica i d'informació, la capacitat d'expressió oral i escrita (sobretot en anglès), la capacitat de síntesi, la presa de decisions per a triar la millor solució comparant diferents propostes i la capacitat per a escollir els paràmetres adequats per a avaluar un sistema. En aquest sentit, el grup de recerca al qual pertanyo ja ha publicat diferents articles sobre el treball que hem anat desenvolupant al llargs d'aquests mesos:

- *Operating System Support for Heterogeneous Multicore Architectures* [16]
- *Adapting ELF to Load Heterogeneous Binaries* [17], presentat al *Third Workshop on Software Tools for MultiCore Systems* (STMCS 2008) [18], celebrat a Boston el 6 d'abril de 2008.

- *OS Paradigms Adaptation to Fit New Architectures* [19], presentat i acceptat al *Fourth Workshop on the Interaction between Operating Systems and Computer Architecture* (WIOSCA 2008) [20], celebrat a Pequín el 22 de juny de 2008. Aquest article resumeix tota la feina que s'ha fet durant tot el Projecte Final de Carrera.

A més d'aquests articles, també ens han acceptat un pòster sobre el treball realitzat al *Annual Technical Conference* USENIX 2008 [21], celebrada a Boston entre el 22 i el 27 de juny de 2008.

Seguint aquesta línia de recerca, el Projecte Final de Carrera no només es centrarà en obtenir uns resultats immediats, sinó també d'aconseguir que el sistema disposi de flexibilitat per a que pugui ésser adaptat a les noves propostes que vagin sorgint, com ja s'ha comentat als objectius del Projecte. També es tindran en compte aspectes de disseny, proposant diferents solucions al problema que es planteja, avaluant les seves avantatges i inconvenients, i decidint la millor alternativa per a resoldre'l, i proposant per on caldria continuar el treball si es decidís fer-ho en un futur.

Tot i això, i com ja s'ha comentat anteriorment, el Projecte està enfocat a arquitectures molt actuals, fent que no sigui fàcil desenvolupar aplicacions ni trobar documentació per a resoldre els possibles dubtes i errors que apareguin. El fet que sigui una arquitectura totalment diferent al que més s'estudia, amb un llenguatge màquina diferent, etc. també dificulta l'assoliment dels objectius.

Per tant, es podria considerar que la realització completa del Projecte és de grans dimensions, i que no és factible en el marc d'un Projecte Final de Carrera. Tanmateix, es podria dividir el Projecte en tres fases ben diferenciades: estudi, proposta i implementació, de les quals les dues primeres sí que són viables per a realitzar en un Projecte Final de Carrera. Per això, nosaltres ens dedicarem principalment a l'estudi i proposta del model, i la seva implementació serà parcial.

4. CONCEPTES GENERALS

En aquest capítol es presenten uns conceptes generals que són útils per a comprendre la resta de la memòria. Si s'està familiaritzat amb l'entorn de treball no és necessària la seva lectura, tot i que és recomanable per entendre millor el que s'explicarà posteriorment.

Com ja s'ha fet a l'anterior capítol, dividirem els conceptes en dos grups principals per a facilitar-ne la comprensió: conceptes generals més relacionats amb la part d'aplicació i de sistema operatiu.

4.1. Aplicació

D'aquesta primera part corresponent al nivell d'aplicació, el més important per analitzar i entendre és el format executable ELF, i com aquest es genera a partir del codi font d'un programa, mitjançant el compilador i l'enllaçador. A continuació expliquem aquests tres conceptes.

4.1.1. Format executable ELF

El format executable ELF (*Executable and Linking Format*) [22] és el més utilitzat actualment a Linux donada la seva flexibilitat i extensibilitat. És un format definit tant per fitxers executables com per llibreries. Inicialment va ser desenvolupat per sistemes de 32 bits, tot i que actualment s'utilitza en entorns molt diversos. És el successor del format *a.out*.

Un fitxer ELF consta de diverses parts:

- Capçalera, on hi ha informació general del fitxer (nombre màgic[6], si és de 32 o 64 bits, tipus d'endian, etc.).
- Una taula de capçalera de programa (*Program Header Table*), que descriu zero o més segments.
- Una taula de capçalera de secció (*Section Header Table*), que descriu zero o més

[6] Conjunt de bytes que identifiquen unívocament un tipus de fitxer (ja sigui executable, imatge, document, etc.). Acostumen a estar a l'inici del fitxer.

seccions.

Els segments contenen informació útil en temps d'execució, mentre que les seccions contenen informació útil a l'hora de compilar i enllaçar. Finalment, comentar que gairebé totes les distribucions Linux disposen d'una eina molt útil, anomenada `readelf`, que extreu tota la informació d'un fitxer ELF.

4.1.2. Compilador

El compilador és el *software* encarregat de traduir un programa escrit en un llenguatge de programació en un altre, generalment codi màquina. Consta de diferents parts, que s'expliquen breument a continuació:

- Anàlisi lèxic: Converteix els caràcters en *tokens* (unitats bàsiques del llenguatge, com ara identificadors o símbols) per facilitar el treball del pàrser, i elimina aspectes irrellevants (espais, comentaris, etc.). Generalment es defineix mitjançant expressions regulars.
- Anàlisi sintàctic: També anomenat pàrser, construeix un arbre que representa l'estructura sintàctica del programa, és a dir, que els *tokens* apareixen en l'ordre correcte. Generalment es descriu mitjançant una gramàtica lliure de context.
- Anàlisi semàntic: És l'encarregat d'afegir informació semàntica a l'arbre, i crea la taula de símbols (amb les variables, noms de funcions, etc.). Fa diverses comprovacions (com ara la de tipus entre variables, les crides es fan a operacions, etc.) per a determinar que el programa és consistent.
- Generació de codi intermedi: A partir de l'arbre construït a l'anàlisi sintàctic i omplert d'informació pel semàntic, es genera un codi intermedi, que podrà ser optimitzat.
- Optimitzacions: En aquesta fase es treballa el codi intermedi, eliminant codi mort, propagant constants, eliminant sub-expressions comunes, realitzant *strength reduction* (per exemple, enlloc de calcular a*64 es pot substituir per a<<6), etc. El fet de realitzar les optimitzacions sobre el codi intermedi fa que no estigui lligat a cap codi màquina en concret.

- Generació de codi màquina: El darrer pas, consistent en traduir l'entrada en el llenguatge de sortida.

4.1.3. Enllaçador

L'enllaçador (també anomenat *linker* [23]) és el *software* encarregat d'ajuntar els diferents fitxers objecte, generats pel compilador, juntament amb les llibreries que aquest necessita, per a generar l'executable final. S'encarrega, entre altres funcions, de resoldre les referències entre els fitxers objecte. És el darrer pas abans d'aconseguir l'executable final.

4.2. Sistema operatiu

El sistema operatiu és el *software* que s'executa en mode privilegiat (pot executar tot el conjunt d'instruccions de la CPU), i s'encarrega de la gestió i administració dels recursos de què disposa la màquina on s'executa. Un resum (a grans trets) de les tasques del sistema operatiu seria:

- Proporcionar mecanismes per a l'execució de tasques per part dels usuaris, com ara les crides a sistema (peticions que fa una aplicació per a demanar algun tipus de servei al sistema operatiu).

- Tractament de les interrupcions (un dispositiu es vol comunicar amb el sistema operatiu) i de les excepcions (l'execució d'un programa ha generat un error que requereix de tractament per part del sistema operatiu).

- Canvi de context

- Gestió de processos (creació i destrucció, assignació de memòria, comunicació entre ells...).

- Gestió de sistemes d'entrada/sortida (perifèrics, discos, xarxa, etc.).

  Tot això s'ha de realitzar amb la màxima seguretat (donat que, com hem dit

anteriorment, el sistema operatiu té accés a tots els recursos de la màquina i un error podria tenir conseqüències no desitjades) i amb la màxima eficiència (ha de ser una capa "transparent" per l'usuari).

4.2.1. Carregador

El carregador, també anomenat *loader* [23], és la part del sistema operatiu encarregada de carregar un procés a memòria. A continuació s'expliquen amb profunditat l'estructura i els passos que segueix Linux per gestionar els formats executables i ser capaç de carregar un fitxer binari a memòria.

Linux disposa d'una llista enllaçada, anomenada `formats`, que conté objectes que representen els formats executables que el sistema operatiu sap carregar a memòria. Els elements d'aquesta llista són de tipus `linux_binfmt`. El darrer element d'aquesta llista sempre és l'objecte que descriu els executables de tipus *script* (que comencen per `#!`). Un exemple d'aquesta llista es pot veure a la Figura 2.

Es poden inserir i eliminar elements de la llista mitjançant les operacions `register_binfmt` i `unregister_binfmt`. Com a curiositat, dir que els elements s'afegeixen pel començament de la llista.

Essencialment, l'objecte `linux_binfmt` consta de 3 punters als següents mètodes:

- `load_binary`: Funció que carrega el binari a memòria, establint un nou entorn pel procés actual llegint la informació de l'executable.
- `load_shlib`: Funció per associar una llibreria dinàmica a un procés ja existent. S'invoca amb la crida a sistema `uselib`.
- `core_dump`: Funció per a escriure el context d'execució d'un procés a un fitxer.

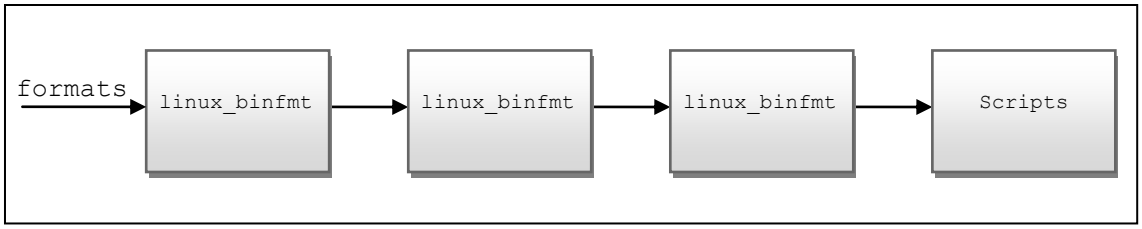


Figura 2: Llista `formats` amb 4 formats executables representats

Quan s'executa una aplicació mitjançant alguna de les crides disponibles a la *libc*[7] (`execl`, `execlp`, `execle`, `execv`, `execvp` o `execve`, que es diferencien entre elles en com s'interpreten els paràmetres) totes acaben invocant a la mateixa crida a sistema: `sys_execve`. Aquesta crida a sistema obté els paràmetres dels registres del processador, salvats abans d'entrar a sistema, i invoca la funció `do_execve`. Aquesta funció crea una estructura de tipus `linux_binprm`, que serà omplerta amb informació del fitxer executable, i realitza altres funcions com ara comprovar que es tenen permisos per executar el fitxer. Finalment, si tot és correcte s'invoca a la funció `search_binary_handler`.

La funció `search_binary_handler` va recorrent la llista `formats`, invocant per cada element de la llista el seu mètode `load_binary`. És a dir, es va intentant carregar el fitxer amb tots els formats reconeguts pel sistema. Comprovant el seu retorn, sap si s'ha pogut carregar el fitxer executable. D'aquesta manera, fins que no s'hagin provat tots els formats executables de la llista o un d'ells hagi pogut carregar satisfactòriament el fitxer, es continuarà intentant carregar el fitxer. Per tant, les funcions han de retornar un error si no saben carregar el fitxer executable.

En el cas dels binaris ELF, la funció associada al mètode `load_binary` s'anomena `load_elf_binary`. Aquesta funció, a grans trets:

1. Comprova el nombre màgic. Si no coincideix amb el nombre màgic de l'ELF retorna un error, indicant que no sap carregar el fitxer.

[7] Llibreria estàndard de C, que proporciona i defineix una interfície per dur a terme les crides a sistema.

2. Llegeix la capçalera per extreure la informació general del fitxer: segments i llibreries dinàmiques que necessita.

3. Obté l'intèrpret del programa, que serà l'encarregat de localitzar les llibreries dinàmiques i carregar-les a memòria.

4. Allibera tots els recursos utilitzats pel procés. Això es fa perquè el procés que ha executat la crida a sistema per carregar el binari és la còpia d'un procés (quan executem a un *shell* un programa intervenen dues crides a sistema, la que serveix per crear un nou procés, còpia del pare, i la que estem comentant ara, que executa pròpiament el programa) i aquests recursos no pertanyen al nou procés.

5. Reserva memòria per la pila d'usuari, el segment de text (és a dir, el codi) i el segment de dades del nou procés.

6. Inicialitza el descriptor de memòria, que conté les adreces d'inici i final del codi i les dades, i l'inici de la pila.

7. Si el programa utilitza llibreries dinàmiques, carrega a memòria l'intèrpret.

8. Invoca la macro `start_thread`, que modifica els registres `%eip` (*Extended Instruction Pointer*) i `%esp` (*Extended Stack Pointer*) de la pila de kernel, fent que valguin el valor de l'intèrpret i el valor de la pila d'usuari, respectivament.

9. Acaba la funció retornant 0.

Al sortir a mode usuari, s'executarà l'intèrpret, que carregarà les llibreries dinàmiques que necessita el programa, actualitzarà totes les referències als símbols de les llibreries, i finalment saltarà a executar el programa.

A la Figura 3 es pot veure un esquema de les crides que s'acaben d'explicar, per facilitar-ne la comprensió.

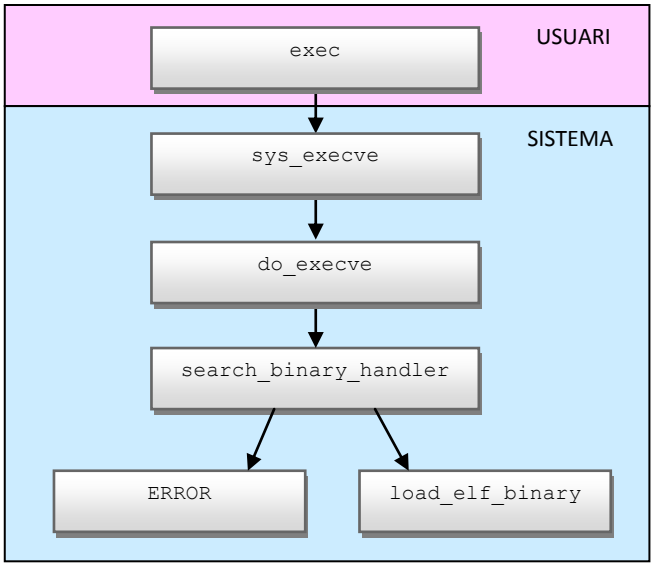


Figura 3: Resum de les crides a l'hora d'executar un programa

4.2.2. Programa, procés i thread

Un altra concepte important és la diferència entre programa, procés i thread, vista des de l'òptica del sistema operatiu.

Un programa és un conjunt d'instruccions, escrites en un llenguatge de programació d'alt nivell, que són traduïdes mitjançant un compilador a codi binari, interpretables directament pel *hardware*.

Un procés no és res més que un programa quan està en execució, amb tota la informació associada al fet que s'està executant, com ara la memòria, els dispositius que utilitza, els fitxers que té oberts, etc.

Finalment, un thread és el flux d'execució d'un procés. Es representa mitjançant l'estat actual del processador, amb els registres i la pila. Un procés, per tant, pot disposar d'un o més threads, que poden estar executant diferents parts del mateix.

Com es pot veure, aquests conceptes són relativament nous. La idea que es tenia fa uns anys de procés (tenir carregada a memòria la imatge d'un binari per a que pugui ésser executat seqüencialment) ha variat lleugerament. Actualment, per les prestacions que ofereix el *hardware*, un mateix procés pot disposar d'un o més threads, o fils d'execució, de

tal manera que es puguin realitzar diferents tasques del mateix procés paral·lelament. El conjunt de tots els threads d'una aplicació, juntament amb la memòria que comparteixen, és el que fa uns anys s'entenia com a procés.

4.2.3. Planificador de tasques

Finalment, el darrer concepte que estudiarem és el planificador de tasques del sistema operatiu, també anomenat *scheduler*. El planificador és un component molt important en un sistema operatiu multiusuari i multitasca, perquè és l'encarregat de garantir que tots els processos que estan preparats per ésser executats s'executin en algun instant, repartint el temps entre tots ells. D'aquesta manera, a l'usuari del sistema li dóna la sensació que totes les tasques s'estan executant alhora, quan en realitat no és així.

En sistemes on només hi ha un microprocessador, només es pot executar un procés a la vegada. A més, els programes no incorporen per sí mateixos cap mecanisme per deixar-se d'executar. Per tant, ha de ser el sistema operatiu l'encarregat de fer aquests canvis entre processos. La missió del planificador és anar triant els processos a executar, seguint algun tipus de política, expulsant al procés que actualment està en execució quan es consideri oportú. Quan es decideixi que s'ha de fer un canvi de procés, els passos a fer són els següents:

- Guardar el context d'execució de la tasca que s'està expulsant de la CPU
- Triar el procés que s'ha d'executar, d'entre tots els processos que estiguin disponibles (per exemple, si un procés està bloquejat esperant alguna dada no és candidat a ocupar la CPU), seguint alguna política de planificació
- Una vegada triat el procés a executar, es restaura el seu context d'execució prèviament guardat

La idoneïtat i correctesa d'un planificador resideix en l'algorisme que tria quin és el proper procés a executar, donat que la resta de tasques a realitzar són força automàtiques. En aquest sentit hi ha molts estudis i propostes fetes, però no entrarem en aquests detalls, ja que no és el propòsit d'aquest Projecte. Si es vol aprofundir en aquest aspecte, existeix

una proposta per l'*scheduler* que també ens ha servit per agafar idees pel nostre projecte i per reflexionar sobre la gestió dels threads d'usuari i els threads de kernel. Aquesta proposta es pot trobar a [24].

5. PRIMERA APROXIMACIÓ

En aquest apartat donarem els primers detalls del Projecte, que ens serviran per establir els requeriments, proposar les solucions per a resoldre el problema plantejat i analitzar i criticar la solució per a obtenir nous requeriments. Així podrem continuar el desenvolupament del Projecte de manera iterativa.

Els primers requeriments que tractarem són els següents:

- Disposar de diferents trossos de codi, compilats per cada ISA que suportem (de manera similar al CESOF), incrustades al mateix fitxer executable.
- Que el carregador detecti la nova informació guardada a l'executable i la emmagatzemi per a la seva posterior utilització, creant els nous objectes a nivell de kernel necessaris per a guardar-la.
- Disposar d'un thread que pugui anar executant el codi, modificant el model d'execució normal d'un thread. El que volem és utilitzar tot el que s'ha implementat per variar una mica el flux d'una aplicació.

En els propers tres apartats donem solució a aquests tres requeriments que ens plantegem. Aquesta serà una primera iteració del Projecte, que ens ajudarà a situar-nos en l'entorn de treball.

5.1. Heterogeneous ELF (HELF)

La nova extensió del format ELF que incorporarà la informació que necessitem emmagatzemar s'anomena *Heterogeneous ELF* (HELF d'aquí en endavant). Està basada en el format ELF, però divideix el codi de diferents ISA's en diferents seccions de codi. Aquest matís el diferencia respecte el CESOF, on la secció que conté l'executable SPE no és de codi, sinó de dades.

En aquesta primera aproximació, el que es divideix són funcions que estan compilades per una ISA en concret. D'aquesta manera, el programador marcaria les característiques d'una funció, i l'executable final reflectiria aquestes característiques a unes noves seccions.

L'altra diferència respecte el format ELF és el nombre màgic, que ens ajudarà a identificar unívocament la nova extensió per a poder dur a terme les accions necessàries a dins el sistema operatiu.

Es pot veure un exemple de codi modificat a la Figura 4, amb tres parts diferents en funció de les característiques del codi, cadascuna representades amb un color diferent. Com ja pot intuir el lector, caldria un compilador adaptat per a que reconegués aquestes marques. Per falta de temps, i perquè no és l'objectiu d'aquest Projecte, aquest compilador no s'ha implementat, sinó que les modificacions del binari s'han fet a mà. Per tant, la nostra proposta és que es marqués amb alguna directiva del compilador, tot i que nosaltres ho generem a mà.

Com es pot apreciar, hi ha una funció `main` que inicialitza unes estructures de dades, crida a una funció que realitza uns càlculs i, finalment, es crida a una altra funció que mostra els resultats. Per tant, podem diferenciar dos tipus de tasques: computació i operacions d'entrada/sortida. Si disposem d'una unitat especialitzada en alguna d'aquestes dues tasques, podem aprofitar aquest fet per a obtenir un major rendiment enviant a executar aquestes tasques a un accelerador.

```
int main (int argc, char *argv[]) {
        int A[10000];
        int B[10000];
        int i = 0;


        while (i<10000) {
                A[11] = i*3;
                i++;
        }


        arrayXint(A,B,29,10000);
        printf("Values after computation:\n");
        printArray(B,10000);
        return 0;
}
```

```
#pragma parallel mmx
void arrayXint (int *A, int *B, int S, int N) {
        int i;
        for (i=0; i<N; i++){
                B[11] = A[11]*S;
        }
}
```

```
#pragma inout
void printArray (int *B, int N) {
        int i;
        for (i=0; i<N; i++){
                printf("Position %d, value %d\n", i, B[11]);
        }
}
```

Figura 4: Exemple de programa amb funcions separades depenent de les seves característiques

A la Figura 5 es presenta l'estructura del binari HELF corresponent a l'exemple de la Figura 4. Com es pot veure, el compilador i l'enllaçador han interpretat les marques afegides pel programador, i han plasmat aquesta informació en unes noves seccions de codi, anomenades `.text.nomFuncio` (on `nomFuncio` és el nom de la funció que representen). Aquest nom podria ser qualsevol altre, però s'utilitza aquesta convenció perquè l'executable sigui més clar per si el volem analitzar.

Les funcions que no contenen cap marca s'agrupen totes a la secció `.text`, de manera idèntica al que fa el format ELF amb el codi.

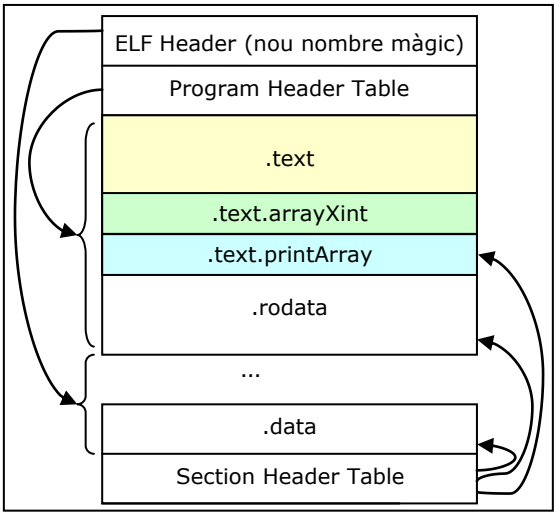


Figura 5: Estructura HELF representant el binari obtingut amb el codi d'exemple de la Figura 4

Si executem aquest binari (separat en funcions i sense canviar el nombre màgic, ja que sinó no es podrà carregar a memòria) en un sistema operatiu sense modificacions, obtindrem el mateix comportament que si tingués una sola secció amb tot el codi, donat que el carregador no interpreta aquestes noves seccions i no se'n pot treure profit. Per aquest motiu és necessari un nou carregador, que s'explica tot seguit.

5.2. Carregador HELF i nous objectes de kernel

Una vegada introduïda l'extensió HELF, anem a tractar el següent requisit, que és el seu carregador. Com ja hem explicat al capítol de conceptes generals, tot format executable necessita el seu carregador específic. Així doncs, cal fer un carregador pels fitxers executables HELF. Com l'extensió HELF està basada en el format ELF, el carregador també ho estarà. Explicarem, doncs, les diferències entre ambdós.

Si provem d'executar un binari HELF com el de la Figura 5 en un sistema operatiu Linux sense cap modificació, obtindríem un error similar al següent:

```
$> ./prova_helf
-bash: ./prova_helf: cannot execute binary file
```

El que succeeix és que no hi ha cap objecte a la llista de formats executables que representi el format HELF, i a l'intentar executar-lo tots retornen un error, que és reportat

a l'usuari. Cal, doncs, crear un nou objecte que representi el format HELF, afegir-lo a la llista de formats reconeguts pel sistema operatiu i implementar els tres mètodes que incorpora. Parlarem únicament del mètode que carrega el binari, donat que és el que ens interessa en aquest moment.

La funció que carrega els executables HELF a memòria s'anomena `load_helf_binary`, per similitud a la seva homòloga pel format ELF `load_elf_binary`. Les dues principals diferències entre aquestes operacions són:

1. Comprovació del nombre màgic. Cada carregador compara el nombre màgic que apareix a la capçalera del binari que s'executa amb el nombre màgic que defineix el seu format. Per tant, l'ELF comprovarà el seu nombre màgic, i el HELF el seu.

2. En el cas del HELF, s'analitzen totes les seccions, i de les que són específiques del format HELF (les que comencen per `.text`.) s'extreu la informació que ens interessa guardar, per emmagatzemar-la en una estructura de dades associada al procés que s'està creant.

La resta de funcions que realitza el carregador ELF es segueixen realitzant, de tal manera que es segueix un camí paral·lel a l'ELF, però amb la possibilitat d'afegir noves funcionalitats sense que aquestes interfereixin en la càrrega normal d'executables ELF. D'aquesta manera queda aïllada la nostra proposta del flux normal d'execució que segueix una aplicació, sent menys intrusius al sistema operatiu. Es podrien solapar les càrregues dels dos binaris, però preferim seguir camins diferents per no interferir en un punt molt utilitzat al sistema operatiu perquè la gran majoria d'aplicacions són de tipus ELF. La Figura 6 reflecteix, resumidament, aquesta situació.

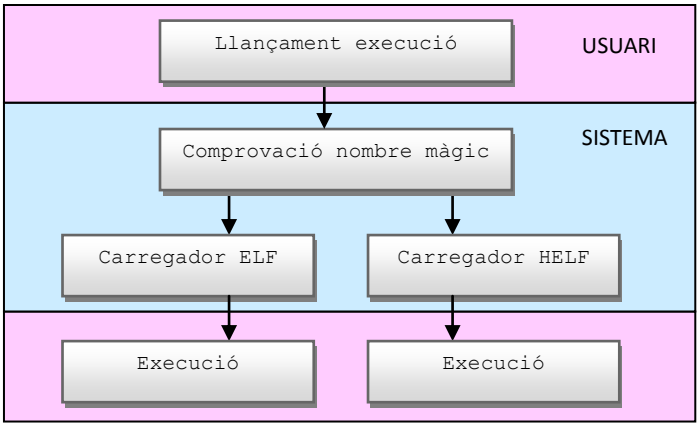


Figura 6: Comparativa entre el flux d'execució d'una aplicació ELF i HELF

Per emmagatzemar la informació que som capaços d'extreure al carregador cal crear unes noves estructures de dades associades al procés, que ens permetin consultar aquesta informació en qualsevol punt de la seva execució sense necessitat de tornar a llegir aquesta informació del binari. Com és informació pròpia de cada procés, la nova informació es guardarà a l'estructura que el sistema operatiu manté per cada procés: la `task_struct`. S'hi han afegit dos nous camps:

- Un enter que identifica si el procés és de tipus HELF o no (per distingir-los).
- Una estructura de dades que conté un conjunt de nodes. Un node representa una secció del binari HELF d'aquest procés, i es qui conté la informació útil de la secció com ara el seu tipus, mida, la seva adreça d'inici, etc. Aquesta estructura de dades s'ha implementat tant en format llista com en format vector, per a comparar les dues alternatives de disseny.

A la Figura 7 es mostren tres figures: una que representa un procés HELF amb l'estructura de nodes implementada amb una llista (a), amb un vector (b) i un procés no HELF amb l'estructura de nodes implementada amb un vector (c). Aquest darrer, com es pot comprovar, té invalidats tots els nodes, ja que no és un procés HELF.

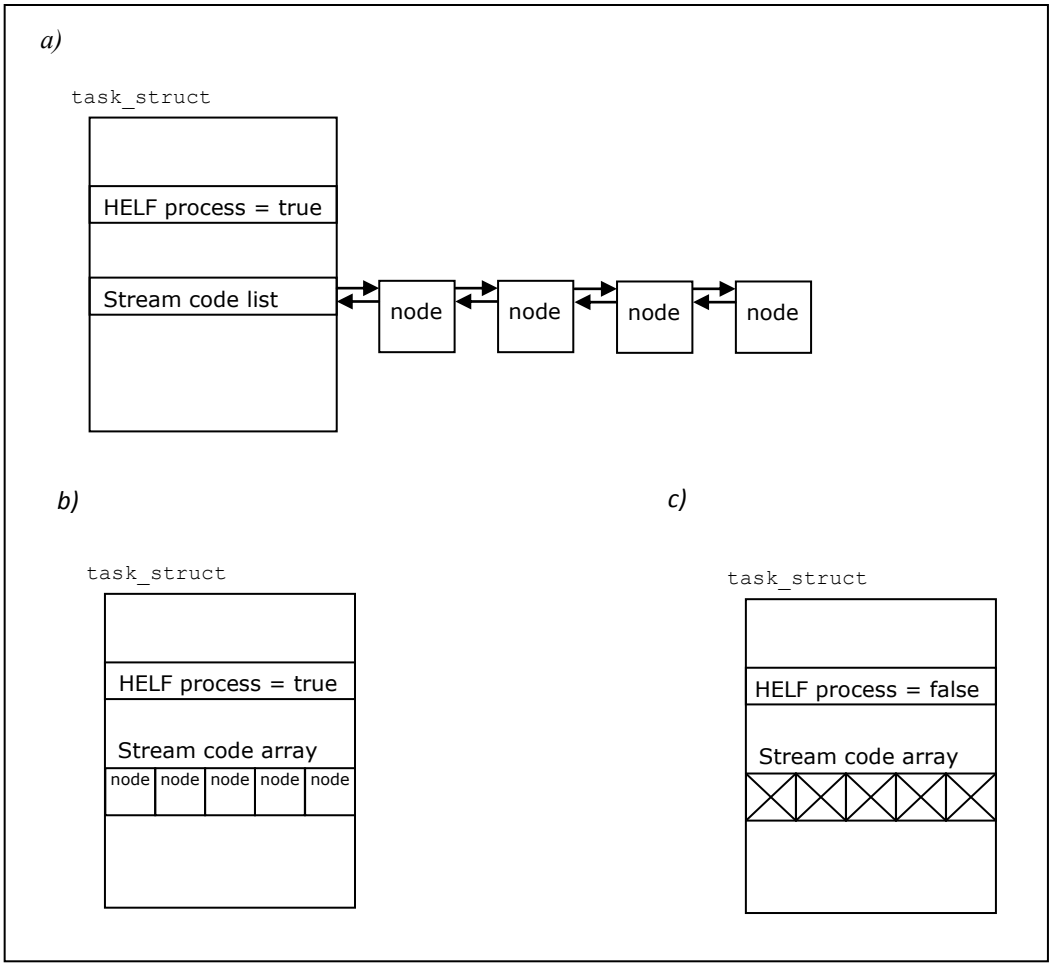


Figura 7: Estructures de dades a nivell de kernel d'un fitxer HELF en format llista (a), en format vector (b) i un fitxer no HELF en format vector (c)

En l'exemple de programa presentat a la Figura 4, el camp que indica si el procés és de tipus HELF seria cert, i hi hauria dos nodes inicialitzats, un per cada nova secció afegida (com s'ha indicat al binari de la Figura 5). El contingut dels nodes dependrien de la secció que representen (la mida de la secció, l'adreça d'inici, etc.).

A la Figura 8 es mostra un esquema d'aquesta situació, perquè quedi més clar com estaria estructurada i enllaçada aquesta informació internament.

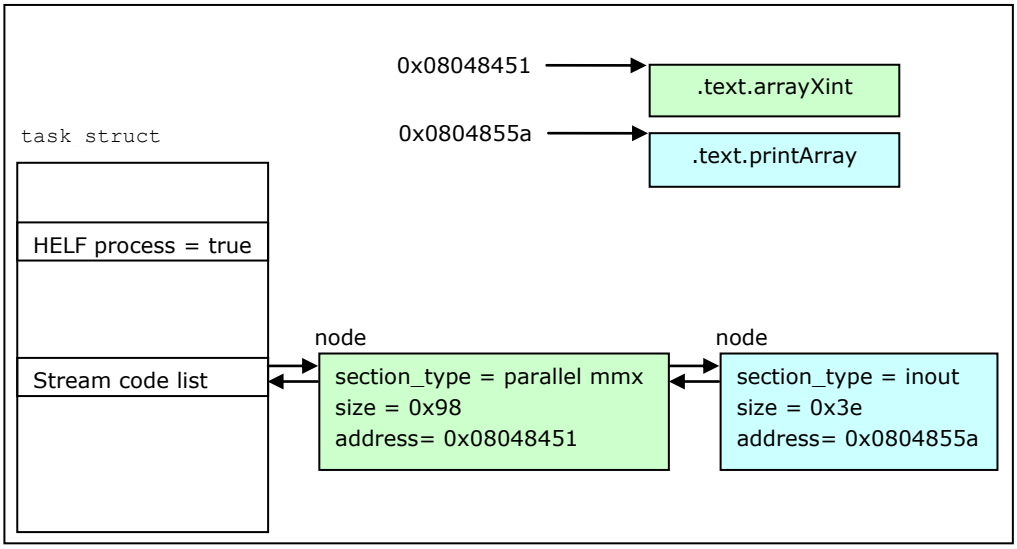


Figura 8: Contingut de les estructures de dades de l'exemple de la Figura 4

5.3. Model d'execució

El darrer apartat que volem estudiar és l'execució utilitzant el que s'ha presentat i implementat fins ara. Ja s'han fet certes modificacions i es podria modificar (ni que fos lleugerament) l'execució d'una aplicació. Tot i això, encara falten pensar certs aspectes relatius als canvis en les unitats d'execució, i el que s'ha fet és treballar i modificar el model d'execució de les aplicacions actuals per utilitzar totes les modificacions fetes fins ara, per que ens anéssim apropant a la idea que un thread vagi executant codi en diferents processadors. Aquesta etapa ens servirà, per tant, com a aprenentatge per estudiar un model d'execució i per proves totes les característiques implementades fins al moment.

Per això es va fer un estudi del funcionament dels *pthreads* i una primera versió de la llibreria que finalment es va acabar implementant amb més funcionalitats i diferent concepció. La llibreria realitzava també noves crides a sistema, i d'aquesta manera es van practicar i dominar aspectes que han acabat sent bàsics la resta del Projecte.

Com a primera aproximació per apropar-nos als objectius del Projecte, el que volem ara és utilitzar tota la infraestructura presentada per a que un *pthread* vagi executant una darrere l'altra un seguit de funcions, utilitzant la informació guardada als nodes. No hi haurà cap salt entre processadors, ni intervenció per part de l'usuari per indicar les funcions a executar, però en servirà com a primera aproximació per aconseguirem els

nostres objectius, i per comprovar que la càrrega del binari HELF és correcta, que la informació s'extreu i es guarda correctament a les estructures de dades associades al procés i que les crides a sistema estan ben implementades. Tot això serà la plataforma per a continuar desenvolupant el Projecte.

En posteriors capítols caldrà tenir en compte els aspectes no trivials que els salts entre processadors suposen (possibilitat de carregar el codi al nou processador si aquest utilitza una memòria local com ara el Cell, etc.) i proporcionar al programador la manera de comunicar a la llibreria la funció a executar.

Com ja hem comentat anteriorment, aquest Projecte s'engloba a dins de la realització d'un Programa de Formació d'Estudiants en Departaments i Instituts. La metodologia pròpia de la recerca és l'anàlisi del problema, i el buscar diferents alternatives per a resoldre'l, per després avaluar les diferents solucions i decidir justificadament quina és la més adient i adequada. És per això que ens plantegem dues possibilitats: una a nivell d'aplicació i una a nivell de sistema operatiu.

Abans, però, també cal resoldre a la següent pregunta.

5.3.1. Perquè utilitzar *pthreads*?

Una de les preguntes que es podria fer el lector és la següent: perquè cal utilitzar *pthreads*? Si el que es vol és anar executant diferents funcions, sense necessitat que s'executin en paral·lel, perquè cal utilitzar *pthreads*? No serveix el mètode "tradicional" d'anar cridant-les?

La raó d'això és que l'execució normal d'una funció, cridada des d'algun lloc del programa principal, no entra a dins del sistema operatiu. Una crida a una funció comença amb una instrucció màquina com ara un `call` (en assemblador i386), que modifica el valor del registre comptador de programa per a executar la funció, i finalitza amb una instrucció com ara un `ret` (també en assemblador i386), que restaura el registre comptador de programa de la pila (guardat al fer el `call`) per a executar la següent instrucció a la crida.

Per tant, aquests salts són transparents al sistema operatiu. En canvi, per la nostra proposta ens interessa prendre el control abans d'executar una funció, i a l'acabar d'executar-la, per a poder decidir aleshores quines accions s'han de dur a terme (carregar el codi al nou processador, transferir-li el control del thread, etc.). En aquest sentit, la llibreria de *pthreads* és la solució més adient per practicar aquests aspectes. A continuació s'explica perquè, presentant breument el funcionament d'aquesta llibreria.

5.3.2. Pthreads

La llibreria de *pthreads* (POSIX Threads) està disponible tant en entorns UNIX com en entorns Windows, i és un estàndard que defineix una API[8] per a la creació i manipulació de fils d'execució.

En concret, l'operació per a crear un *pthread* (`pthread_create`), que té com a un dels paràmetres la funció que volem que executi el *pthread*, s'implementa internament mitjançant una crida a sistema `sys_clone`, que també és cridada a l'hora de fer un `fork`. Per tant, l'únic que fa aquesta crida és crear un nou procés com a còpia del procés que la crida.

Quan passa, llavors, a executar la funció que li hem dit que executi? Al sortir de la crida a sistema (ja tornem a estar en mode usuari), sabem que el seu retorn és:

- el PID del fill, al pare.
- un 0, al fill.

La llibreria de *pthreads* utilitza això per a saber si es tracta del pare o del fill i, en el cas de fill, prepara els paràmetres de la funció (guardant-los a la pila en el cas de l'arquitectura x86) i executa un `call` a l'adreça que li hem passat com a paràmetre.

A la Figura 9 es presenta un petit esquema d'aquesta situació, mostrant la crida que

[8] *Application Programming Interface*, un conjunt de rutines que s'ofereixen als programes per a poder ser utilitzats

fa el programador a la part d'aplicació, el salt a la llibreria, l'entrada al sistema operatiu (*trap*, en color vermell) i la crida a la funció especificada pel programador.

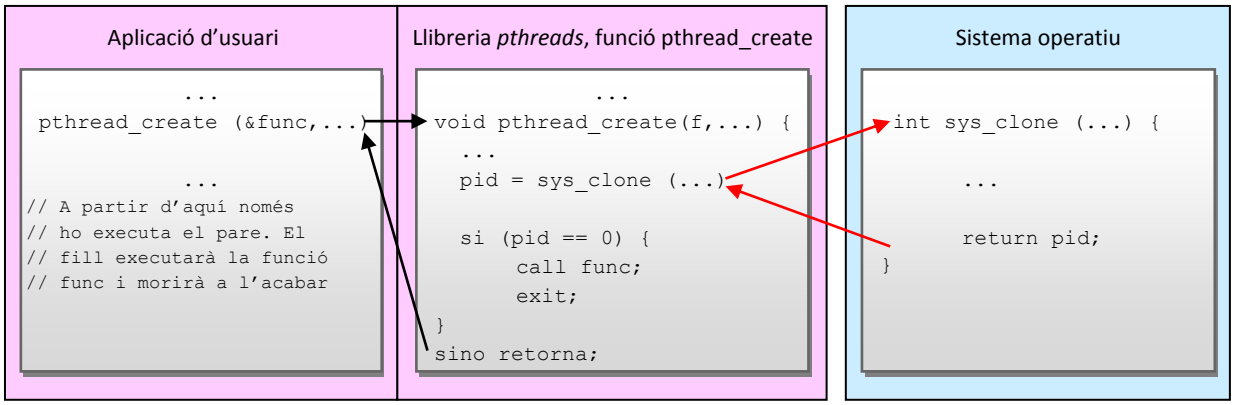


Figura 9: Resum de les crides involucrades, des del nivell d'aplicació fins al de sistema operatiu, en la creació d'un *pthread*

Per altra banda, l'operació per a destruir un *pthread* (`pthread_exit`) l'ha d'inserir el programador al final de la funció que vol que executi el *pthread*, i el que fa és una crida a sistema `_exit`, que s'encarrega de finalitzar un thread, independentment de la resta de threads que hi hagi al grup de la víctima, a diferència de la crida a sistema `exit`, que finalitza tot un procés amb el seu grup de threads, és a dir, tota una aplicació multithreaded. Aquesta darrera és la que insereix automàticament el compilador al final d'un programa `main`.

Per tant, l'avantatge d'utilitzar *pthreads* és que ens garanteixen que abans d'executar la funció es farà una crida a sistema, i al finalitzar també, i podrem prendre les decisions que ens interessin en aquests punts. Això és precisament el que a nosaltres ens interessa: entrar al sistema operatiu abans i després d'executar una determinada funció. És per això que farem un estudi previ del model d'execució amb aquesta llibreria, per anar agafant idees.

5.3.3. Primera opció: llibreria hpthread

La primera alternativa implementada ha estat la de disposar d'una llibreria a nivell d'usuari que realitzés el que ens proposem. Aquesta llibreria l'anomenarem *hpthread* (*Heterogeneous pthread*), per estar basada en la llibreria *pthread*. Aquesta llibreria anirà demanant al sistema operatiu si hi ha més funcions per executar, i si és així les anirà

executant una darrera l'altra. En concret, la llibreria implementa un subconjunt de les disponibles a la llibreria *pthread* modificant el seu comportament pel desitjat en el nostre cas.

La funció per crear un *hpthread* no necessita cap paràmetre, ja que anirà executant les funcions que, a l'haver estat separades en diferents seccions, es troben a les estructures de dades del procés. Això es realitza mitjançant una crida a sistema implementada: `get_next_helf_section`, que va recorrent l'estructura de dades (ja sigui la llista o el vector) i retorna l'adreça de la següent funció a executar o 0 si no s'ha d'executar cap més funció.

Una vegada s'hagi executat la primera funció, el programador haurà afegit al final de la funció la crida `hpthread_exit`, que comprovarà si hi ha més seccions a executar. Si n'hi ha més, farà un `call` de manera idèntica a com es fa en la llibreria de *pthreads*. Aquesta funció, a la vegada, tindrà com a última instrucció `hpthread_exit`, amb la qual cosa s'aniran executant totes les funcions iterativament. Quan no n'hi hagi més, el *hptread* finalitzarà.

Finalment, l'operació `hpthread_join` té un comportament idèntic a la seva equivalent en *pthreads*, que és bloquejar a qui la crida fins que el thread finalitzi la seva execució.

A la Taula 1 es mostra un resum de les operacions de la llibreria, juntament amb una petita descripció de les seves funcionalitats, per resumir tot el que s'ha explicat sobre el seu funcionament.

Operació	Funcionalitats
`hpthread_create`	Quan el procés l'invoca, demana al sistema operatiu (mitjançant una crida a sistema) quina és la primera funció per a executar. Si n'hi ha alguna, es crea un *pthread* per a que l'executi. Quan acaba d'executar la funció, el *pthread* ha d'invocar a `hpthread_exit` (inserida pel programador al final de la funció). Si no hi ha cap funció per a executar, retorna un error.
`hpthread_exit`	Igual que `hpthread_create`, demana al sistema operatiu si hi ha més funcions per a executar. Si és així, el *pthread* salta a l'adreça de memòria on comença la nova funció. Al sortir de sistema i retornar a usuari, es trobarà executant aquella funció. Quan acabi, haurà d'invocar una altra vegada a `hpthread_exit`, i de manera iterativa amb totes les funcions que s'hagin separat en seccions. Quan ja no n'hi hagi cap més, el *pthread* morirà invocant la funció `pthread_exit`.
`hpthread_join`	Amb aquesta funció el procés principal es bloquejarà fins que el *pthread* creat acabi. Si no hagués creat cap *pthread* anteriorment mitjançant `hpthread_create`, aquesta funció no fa res.

Taula 1: Resum de les operacions disponibles a la llibreria `hpthread`

Com es pot veure, aquest funcionament no té en compte ni els paràmetres de la funció i el retorn. Són coses que, ara per ara, obviarem. Només ens interessa modificar el flux d'execució, sense preocupar-nos de les dades. També cal destacar que la llibreria només permetia l'execució d'un sol thread, per simplicitat. Tot i això, adaptar-la per a múltiples threads seria tant senzill com disposar de més identificadors de thread, i en cada operació discriminar quin d'ells l'ha cridat i actuar en conseqüència.

El comportament de la llibreria ha estat el desitjat. És a dir, un programa `main` l'únic que havia de fer era cridar a la funció per crear un *hpthread*, i després esperar que finalitzés. Aquest anava obtenint les adreces de les funcions que prèviament s'havien separat en diferents seccions, i les anava executant correctament (com ja hem dit, sense paràmetres d'entrada ni sortida). Finalment, quan s'acabaven d'executar totes, el thread moria, i el programa `main` finalitzava.

El principal avantatge d'aquesta solució és que a l'implementar-la a nivell d'usuari, no cal recompilar el kernel cada cop que es vol modificar alguna cosa. Una vegada implementada la crida a sistema, tots els possibles errors d'implementació provoquen com a màxim l'aturada del programa, però no del sistema sencer. El principal desavantatge és

que l'usuari no té cap tipus de control sobre el flux d'execució de l'aplicació.

Per tant, és una solució que ens ha servit pel que preteníem en aquest capítol (comprovar que tot el que s'ha fet fins ara és correcte i practicar amb un model d'execució per anar canviant el flux d'una aplicació), però en cap cas serveix com a solució final.

5.3.4. Segona opció: modificació del sistema operatiu

La segona alternativa que ens vam plantejar va ser modificar directament el sistema operatiu. Per fer això, cal conèixer perfectament els passos que es realitzen en la creació i destrucció dels threads. A continuació els expliquem amb detall. Si li interessa al lector ampliar la informació que es presenta, es recomana consultar a [25].

Creació d'un thread

Com ja s'ha comentat quan s'ha explicat el cicle de vida d'un *pthread*, la crida a sistema invocada al crear un thread és la crida `sys_clone`. Aquesta crida està implementada al fitxer `arch/i386/kernel/fork.c`. A continuació s'explica breument els passos que segueix.

La capçalera de la funció és la següent:

```
asmlinkage int sys_clone(struct pt_regs regs)
```

La paraula clau `asmlinkage` indica al compilador que no busqui els paràmetres als registres del processador (una optimització que s'acostuma a fer quan hi ha pocs arguments), sinó que els busqui a la pila.

El paràmetre és una estructura que conté els registres del microprocessador.

El codi de la crida és força simple. Només extreu la informació que es necessita passar a la funció `do_fork` dels registres passats com a paràmetre (*flags* que determinen què s'ha de copiar i què no, l'adreça de la pila del nou procés, els registres que rep com a paràmetre el `clone`, etc.) i fa la crida a la funció `do_fork`.

La funció `do_fork` reserva memòria per a una estructura de tipus `pid`, que identificarà al nou procés, i li passa com a un dels paràmetres a la funció `copy_process`, que retornarà un punter a la nova `task_struct` creada. Posteriorment, depenent si s'ha indicat amb els *flags* que el nou procés es posi a executar o "neixi" aturat, es despertarà mitjançant una crida a `wake_up` o es posarà el seu estat a `TASK_STOPPED`.

La funció `copy_process` comença duplicant sencera la `task_struct` actual (que és el pare del nou procés a crear), després posa a 0 els diferents camps de la `task_struct` propis d'un procés que no s'han d'heretar (com ara comptadors d'entrada sortida de bytes llegits i escrits a dispositius), o resetejant llistes pròpies del procés. És una funció força extensa, i no ens allargarem en el seu contingut. Només destaquem una de les moltes crides que fa, anomenada `copy_thread`, perquè ens és d'interès per a nosaltres.

La funció `copy_thread` és la que inicialitza els registres de la pila de kernel del procés fill. En concret, modifica el registre `%eax` fent que valgui 0 (d'aquesta manera, quan retornem a usuari el fill hi tindrà aquest valor, com ja hem dit anteriorment que succeeix), el registre `%esp` (*Extended Stack Pointer*) fent que valgui l'adreça de la pila que hem anat passant en totes les funcions, i el registre `%eip` (*Extended Instruction Pointer*) fent que valgui una funció anomenada `ret_from_fork`. Aquesta funció crida alhora a `ret_from_syscall`, que serveix per restaurar tots els registres de la pila de kernel (l'estat del procés) i força a la CPU a canviar de mode, sortint de sistema a usuari, mitjançant la instrucció màquina `iret`.

Destrucció d'un thread

La crida a sistema invocada al destruir un thread és la crida `_exit`. Aquesta crida està implementada al fitxer `kernel/exit.c`. Aquesta crida no és tant complicada, ni molt menys, que la que crea el thread.

Com s'ha comentat en l'apartat anterior, existeixen dues crides a sistema per finalitzar una aplicació en mode usuari: una finalitza tot el grup de threads d'una aplicació (la crida a sistema `exit_group`, que és la invocada per la funció `exit` que afegeix el compilador automàticament quan s'acaba un programa), i l'altra finalitza un sol thread (la

crida a sistema `_exit`). Explicarem aquesta segona, ja que l'únic que fa és cridar a `do_exit` que és el que acaba fent també la crida `exit_group`.

La funció `do_exit` comprova primerament que el procés que la crida no sigui ni el procés *init*, ni el procés *idle*, ni el que tracta les interrupcions, donat que cap d'ells hauria de morir. Si és així, genera un *panic*, que indica un problema greu a dins del kernel.

A continuació, marca el procés com que s'està eliminant, i comença a eliminar tota la informació associada al thread, com ara la memòria, semàfors, fitxers oberts, mòduls, etc. Per últim, allibera la `task_struct` associada al thread, i executa un canvi de context. Una vegada es triï un nou procés per a ocupar la CPU, podem dir que el procés ha mort.

Una vegada coneixem quines accions es realitzen al crear i al destruir un thread, comentar que vam intentar fer que no fes falta crear un thread, sinó que a l'acabar d'executar una funció i encara hi hagués seccions per executar, el thread s'adormís. Al tornar a entrar al sistema operatiu per crear un nou thread, detectàvem que teníem un thread adormit, i l'aprofitàvem sense haver de crear-ne un de nou.

Tot i això, vam veure que no era exactament el que ens interessava pel Projecte, fet pel qual vam abandonar aquesta proposta a nivell de sistema operatiu.

5.4. Mesures i resultats

En aquest darrer apartat volem mesurar la sobrecàrrega introduïda a nivell de kernel en les modificacions explicades a l'apartat 5.2, tant en temps com en memòria, per les dues alternatives proposades (llista i vector). Així podrem analitzar les avantatges i inconvenients de les dues propostes.

En aquesta primera aproximació hem proposat afegir uns nous objectes que ens serviran perquè el sistema operatiu pugui planificar l'execució de tasques entre els diferents processadors que disposi la màquina, tot i que no hem aprofundit gaire en la part d'execució. Per tant, només avaluem la plataforma per a que els objectius presentats a

l'apartat 1.1 d'aquest document es puguin realitzar.

5.4.1. Mesures de temps

A continuació presentem les gràfiques on es comparen els temps d'execució de les diferents funcions implicades (Figura 10 i Figura 11):

- Carregador: Es miren les seccions i s'omplen les estructures de dades a on es guarda la informació.
- Còpia de procés: Es copien les noves estructures de dades perquè les hereti el fill.

Es comparen entre sí binaris en format ELF, en format HELF sense seccions addicionals i amb 6 seccions addicionals. Les mesures s'han pres a l'inici i al final d'aquestes funcions, amb la instrucció màquina `rdtsc`[9], que retorna el nombre de cicles transcorreguts des que s'ha iniciat la màquina. L'entorn d'execució ha estat controlat, aïllant al màxim l'execució de l'aplicació, de tal manera que tots els recursos de la màquina estiguessin disponibles, i no hi hagués distorsió als resultats obtinguts. Les proves entre binaris s'han fet en igualtat de condicions per a que els resultats siguin equiparables. S'han realitzat tantes execucions com ha estat necessari per apreciar que els resultats eren prou estables i el seu resultat no variava excessivament (entre cinc i deu execucions).

[9] A l'assemblador x86, la instrucció `rdtsc` és un codi mnemotècnic per *Read Time Stamp Counter*.

Carregador

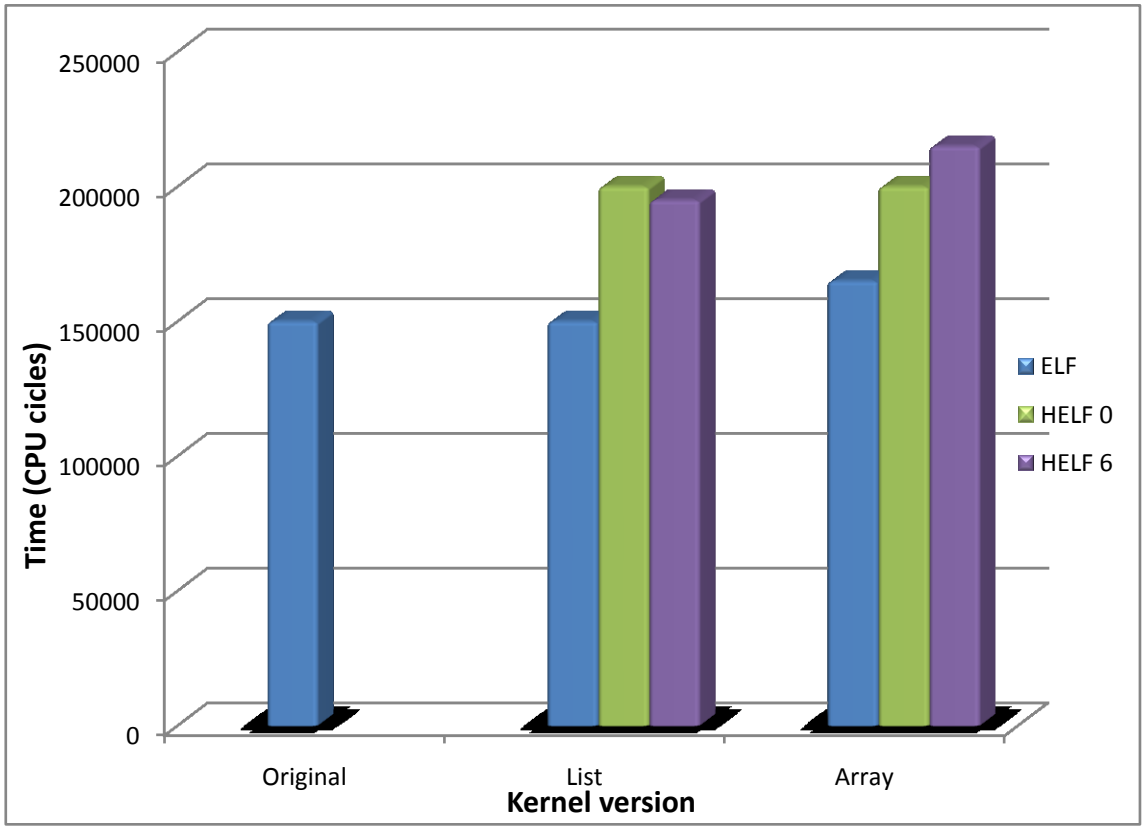


Figura 10: Temps de càrrega d'un binari amb les dues alternatives de disseny: llista i vector

Amb el binari ELF es pot comprovar com amb el carregador normal de Linux (`load_elf_binary`), el kernel original i el de la llista tenen uns temps similars (inicialitzar la llista de nodes té un cost insignificant, ja que és modificar dues variables de tipus punter) mentre que la proposta del vector tarda més pel cost d'inicialització del vector (cost lineal en funció del nombre d'elements del vector).

Amb el binari HELF sense seccions addicionals (HELF 0), el carregador (`load_helf_binary`) funciona igual en les dues propostes (ja que no intervenen les estructures de dades pel fet de no tenir seccions addicionals), mentre que amb el que té 6 noves seccions (HELF 6) tarda més la versió del vector, probablement perquè cal anar comprovant si ja s'han tractat el nombre màxim de seccions que es poden guardar, problema que no té la versió de llista per funcionar amb memòria dinàmica.

L'increment de temps és d'aproximadament 50.000 cicles de CPU, que traduït a

segons en el processador on s'han dut a terme les proves (freqüència de rellotge de 1,86 GHz) serien uns 27 µs.

Còpia de procés

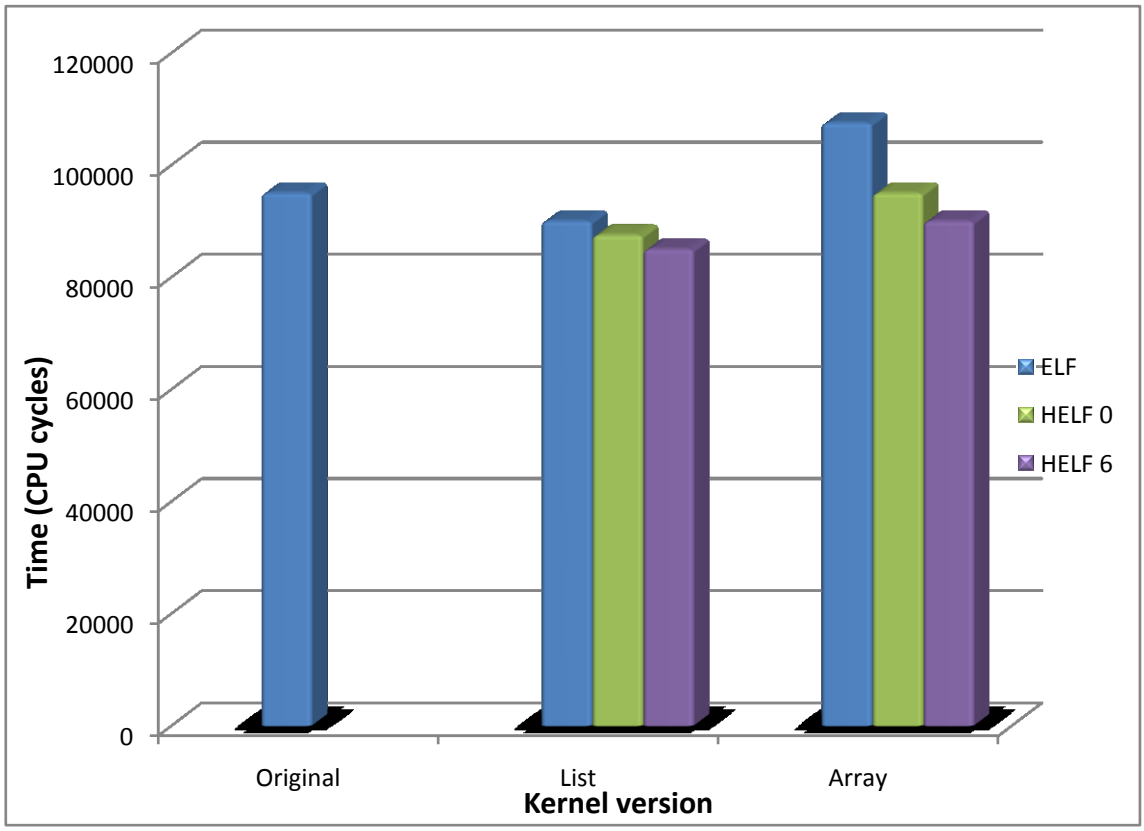


Figura 11: Temps de càrrega d'un binari amb les dues alternatives de disseny: llista i vector

Quant a la còpia del procés, es pot veure com tarda més la versió en format vector, donat que s'han de copiar totes les posicions del vector encara que tinguin valors invàlids. En canvi, a la llista es copien només els elements que hi ha (que en un cas mig seran menys que el màxim). També s'aprecia com aquí tampoc afecta excessivament el nombre de seccions del binari.

L'increment en aquesta operació és d'aproximadament 15.000 cicles de CPU, que serien uns 8 µs en les mateixes condicions que les proves anteriors.

5.4.2. Mesures de memòria

Parlem ara de l'augment de memòria que comporta la nostra proposta. Tot i que sabem que la `task_struct` té una mida fixada en temps de compilació del kernel (8 KB, per exemple) i que encara que hi afegim nous camps seguirà ocupant això (a no ser que ens excedíssim d'aquesta mida, cosa que portaria greus problemes), creiem que és bo avaluar aquest afegit, entre d'altres coses perquè aquest espai és compartit entre els camps de la `task_struct` i la pila, fent que quedi menys espai disponible per aquesta (si poséssim molts camps i envaíssim l'espai de pila també hi hauria comportaments no desitjats).

La `task_struct` del kernel utilitzat per fer aquestes proves ocupa, sense afegir-hi les nostres modificacions, 1392 bytes.

Si ens decidim per utilitzar com a estructura de dades la llista, estem afegim un enter (que ens diu si és un procés HELF o no) i dos punters (que apunten a l'inici i final de la llista). Per tant, són 12 bytes. Cada node de la llista, a més, ocupa 20 bytes (dos punters per enllaçar i 4 enters per guardar la informació que ens interessa).

En canvi, amb el vector, afegim el mateix enter per indicar si és un procés HELF, i un vector de 10 elements (hem establert que el binari tindrà un màxim de 10 seccions afegides pel format HELF), on cada element ocupa 16 bytes (els 4 enters comentats en l'opció anterior).

Per tant, suposant les mateixes restriccions per les dues propostes (un màxim de 10 nodes), el vector afegeix una quantitat fixa (164 bytes), mentre que la llista oscil·la entre un mínim de 12 bytes (llista buida) i un màxim de 212 bytes (llista amb 10 nodes). És a dir, amb poques seccions ocupa menys espai la llista, però amb el nombre màxim de seccions suportades ocupa menys el vector. En un cas intermedi ocupen una mida similar.

Amb totes aquestes proves, podem concloure que les dues alternatives de disseny són similars, tot i que és lleugerament més eficaç la versió en forma de llista. L'avantatge del vector és que no utilitza memòria dinàmica, tot l'espai ja està reservat, i l'accés als elements és més còmode que no en la llista.

5.5. Conclusions

Presentem en aquest darrer apartat les conclusions a les que hem arribat en aquesta primera aproximació del Projecte, per veure com està actualment la proposta.

En primer lloc, hem definit una extensió al format executable ELF, anomenat HELF (*Heterogeneous ELF*), que ens ofereix la possibilitat de disposar de noves seccions de codi que contenen codi de funcions, compilades per diferents ISA's. Un inconvenient d'aquesta extensió és que es pot disparar el nombre de seccions si hi hagués moltes funcions compilades per diferents ISA's.

En segon lloc, hem introduït modificacions al sistema operatiu per manegar aquesta nova extensió. En concret, un nou carregador pel format HELF, que ens permetrà extreure la informació continguda en un binari, i uns nous objectes de kernel associats al procés, per emmagatzemar aquesta informació.

En tercer lloc, hem modificat lleugerament un model d'execució, canviant el flux de l'aplicació utilitzant una llibreria, que tot i no ser exactament el que necessitem ens ha permès provar les modificacions introduïdes i conèixer amb més detall el flux d'execució d'una aplicació.

Finalment, hem mesurat la proposta a nivell de sistema operatiu, tant en temps com en memòria utilitzada, amb dues alternatives de disseny proposades, per analitzar si la penalització introduïda és molt elevada, i hem vist que el cost afegit és assumible.

6. PROPOSTA PER ARQUITECTURES HOMOGÈNIES

En aquest capítol es finalitza la implementació del sistema en una arquitectura multiprocessador homogènia, millorant els aspectes més fluixos de la proposta presentada al capítol anterior, i adreçant els problemes que no s'han plantejat amb profunditat fins ara com és el model d'execució.

Les propostes presentades fins ara es van redactar en l'article "*Adapting ELF to load heterogeneous binaries*" [17], que es va enviar al *Third Workshop on Software Tools for MultiCore Systems (STMCS 2008)* [18]. Tot i que no el van acceptar, el fet d'enviar-ho a aquest *workshop* ens va servir per rebre un *feedback* molt interessant, que ens va ajudar molt a millorar la nostra proposta. Rebre comentaris de gent experta en arquitectura de computadors, que no està tant ficada en el Projecte i veu la proposta des d'una altra perspectiva, ajuda molt a trobar els aspectes més fluixos que cal millorar i a aprendre a comunicar de manera clara i ordenada les idees que es tenen. Tot això són aspectes fonamentals per dur a terme un treball de recerca com el que s'està fent, i un dels principals objectius del Programa de Formació que es duu a terme conjuntament al Projecte Final de Carrera.

En concret, vam rebre dos comentaris de molt d'interès, un d'un revisor que acceptava l'article i comentava els punts a millorar, i l'altre el rebutjava donant els seus motius. A continuació presentem un resum dels comentaris rebuts, dividint-los en comentaris positius i en comentaris crítics.

<u>Comentaris positius</u>

- Donem una nova percepció de l'execució de codi en plataformes heterogènies.
- Proporcionem un bon primer pas en la construcció d'un entorn d'execució més flexible.
- Mostrem la necessitat de migració dels threads a través de diverses unitats d'execució.
- Plantegem noves qüestions sobre la interfície de comunicació entre el codi del programa i la plataforma i la migració de tasques (per exemple, com es gestiona la pila d'execució d'una ISA a una altra?).

Comentaris crítics, millores proposades

- No aclarim perquè el nom de les noves seccions és `.text.nomFunció`, quan podria ser per exemple `.text.nomISA`, i d'aquesta manera ajuntar tot el codi compilat per a una mateixa ISA en una sola secció.
- El model d'execució no és gaire convincent, i no tractem el paral·lelisme de les aplicacions.

Amb tota aquesta informació, podem establir els requeriments (de manera similar a com hem fet amb el capítol anterior) que ens guiaran per finalitzar la implementació en arquitectures multiprocessador homogènies. Aquest serà el pas previ per portar la nostra proposta a un entorn heterogeni, assolint així els objectius globals del Projecte. Els requeriments d'aquesta segona fase són els següents:

- Disposar d'un binari que agrupi tot el codi compilat per a una mateixa ISA a dins d'una mateixa secció de codi, seguint el format HELF.
- Ampliar les capacitats de la llibreria, fent que proporcioni una interfície còmoda i compatible tant amb les arquitectures existents com amb les noves, i que faciliti la introducció de paral·lelisme en les aplicacions.
- Adaptar les modificacions introduïdes a nivell de sistema operatiu per donar suport a les noves funcionalitats i necessitats de la part d'aplicació i de la llibreria.

Tot això ens servirà per ampliar i perfeccionar el flux d'execució presentat al capítol anterior, on el programador no tenia control sobre quines funcions s'anaven executant, sinó que tota la responsabilitat la tenia la llibreria i el sistema operatiu, fent que la proposta no fos prou adient. Ara volem que el programador pugui indicar:

1. Quina funció s'ha d'executar.
2. Quins paràmetres té aquesta funció.
3. On s'ha de guardar el retorn d'aquesta funció.
4. A quina arquitectura s'ha de saltar.

Per tant, el que volem és oferir una abstracció perquè el programador indiqui d'una

manera genèrica on vol executar un codi però, com ja hem dit, el model d'execució que seguirem serà el mateix que si ho féssim de la manera habitual.

Com també es pot veure, la manera de treballar amb la llibreria és similar a la de la llibreria *pthreads*, un dels objectius que ens vam proposar al començar el Projecte per facilitar-ne l'ús als programadors.

A continuació explicarem les decisions que s'han pres per donar resposta a aquests requeriments i aconseguir aquesta abstracció. Després analitzarem la solució proposada amb diferents proves i resultats, i conclourem l'apartat amb unes conclusions sobre tot el que s'ha desenvolupat fins ara.

6.1. Millores del format HELF

En aquest apartat explicarem com hem agrupat les diferents seccions de codi del binari en funció de la ISA per la qual estan compilades, creant una secció de codi que representi i inclogui tot el codi de cada ISA.

Cal dir que en alguns apartats d'aquest capítol parlem indistintament d'ISA i arquitectura, ja que entenem que aquestes tenen una relació unària, és a dir, una ISA és interpretada per una sola arquitectura, i una arquitectura només pot interpretar una sola ISA. Cada secció del binari representa una determinada ISA, que la interpretarà una arquitectura en concret.

Primerament, cal decidir quines arquitectures suportarà el format HELF, i com s'anomenaran les seccions de codi que continguin el codi d'aquestes arquitectures. Per començar, hem definit set arquitectures diferents, tot i que el sistema es podria ampliar si es necessités. A la Taula 2 es mostra una llista amb nom d'aquestes set arquitectures, el nom de la seva corresponent secció en un binari HELF (suposem que sempre comença per `.text.` , ja que la secció principal de codi del binari s'anomena `.text`, i s'hi afegeix el nom indicat a la taula), i un codi per a identificar-les numèricament quan sigui necessari. Per facilitar el seu ús, existiran constants que defineixin aquests codis (la constant serà

`ARCH_NOMSECCIO`, on `NOMSECCIO` és el nom de la secció, definit a la taula, escrit amb majúscules).

Arquitectura	Nom de la secció	Codi numèric
ARM	arm	1
FPGA	fpg	2
PowerPC	ppc	4
SPE	spe	8
Ultra Sparc	usp	16
x86 - 32 bits	x32	32
x86 - 64 bits	x64	64

Taula 2: Arquitectures suportades, nom de la secció associada i codi numèric que les representa

El mètode per a marcar el codi font segons l'arquitectura per la qual s'ha de compilar hauria de ser marcant d'alguna manera les funcions. En aquest cas, la proposta és la mateixa que l'exemple que hem presentat a la primera etapa: introduir la directiva `#pragma nomArq` abans de la capçalera de les funcions que es vulguin executar en un processador determinat, on `nomArq` pot ser `arm`, `fpg`, `ppc`, `spe`, `usp`, `x32` o `x64`. Com ja hem dit, per a poder realitzar això necessitem un compilador adaptat, que sigui capaç d'entendre les directives i compilar el codi adientment, generant el codi màquina per l'arquitectura indicada. Nosaltres, per tant, ho farem manualment.

Totes les funcions que tinguin el mateix nom d'arquitectura s'ajuntaran en la mateixa secció, que representarà tot el codi d'aquella arquitectura. A més, si el compilador ho suportés, es podria indicar més d'una arquitectura en un mateix `#pragma`. D'aquesta manera, aconseguiríem tenir una mateixa funció compilada per a diverses ISA's i es podria decidir en temps d'execució, per exemple, quina versió de la funció executar en funció de diversos paràmetres, com podria ser la càrrega dels diferents processadors o acceleradors de la màquina o el rendiment que s'obté en cada processador (caldria fer una fase d'aprenentatge prèvia per disposar d'aquesta informació).

A continuació, il·lustrem un exemple, per a fer més entenedora l'explicació d'aquest apartat. El codi d'exemple s'ha extret del *benchmark* Scimark2, i s'ha adaptat a la nostra

proposta. La Figura 12 mostra la separació del codi font, marcat amb la directiva `#pragma`.

```
int main(int argc, char *argv[]) {
    double res[6] = {0.0};
    res[1] = FFT(N, mintime, R);
    res[2] = SOR (N, mintime, R);
    res[3] = MonteCarlo (mintime, R);
    res[4] = SparseMatMul (N, nz, mintime, R);
    res[5] = LU (N, mintime, R);
    res[0] = (res[1]+res[2]+res[3]+res[4]+res[5])/5;
    print_out_results(&res);
    return 0;
}
```

```
#pragma x64
double FFT(int N, double mintime, Random R) {...}

#pragma x64
double SOR(int N, double mintime, Random R) {...}
```

```
#pragma spe
double MonteCarlo(int N, int nz, double mintime, Random R) {...}

#pragma spe
double SparseMatMul(struct SMM_params * p) {...}
```

```
double LU(int N, double mintime, Random R) {...}
```

Figura 12: Scimark2 separat en funcions per ser executades en unitats especialitzades

Com podem veure, la funció `main` i la funció `LU` no tenen cap marca, fet pel qual s'agruparan a la secció genèrica de codi `.text`. Les funcions `FFT` i `SOR` es compilaran per a una arquitectura x86 de 64 bits, i s'inclouran dins la secció `.text.x64`. Finalment, les funcions `MonteCarlo` i `SparseMatMul` es compilaran per a ser executades en un SPE, i s'encapsularan dins la secció `.text.spe`. Els diferents colors representen aquest fet, tal i com es va fer amb la Figura 4.

A la Figura 13 es mostra com quedaria dividit el binari HELF del programa d'exemple.

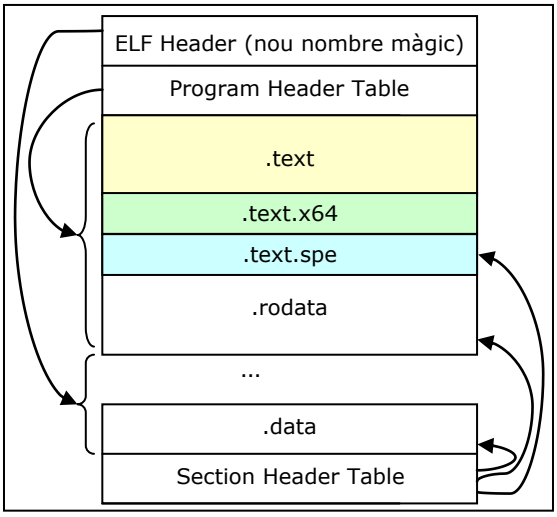


Figura 13: Estructura HELF representant el binari obtingut amb el codi d'exemple de la Figura 12

6.2. Llibreria HELF

Una vegada presentades les modificacions fetes a la part de l'aplicació, anem a veure els canvis fets a la llibreria, tenint en compte els requisits que ens hem proposat referents a la llibreria.

En primer lloc, i a diferència de la llibreria del capítol anterior, hem eliminat la utilització dels *pthreads*. La nostra llibreria, per tant, el que ofereix a l'usuari són possibilitats d'explotar l'heterogeneïtat de les unitats d'execució d'una màquina; que un thread pugui anar canviant de processador o accelerador. En aquest sentit, deixem al gust del programador l'elecció del tipus de paral·lelisme que vulgui per a la seva aplicació, si és que en vol, ja que la nostra llibreria (complint un dels requisits que ens vam marcar als objectius) és compatible amb altres llibreries que podrien afegir paral·lelisme.

Per tant, l'objectiu de la llibreria no és oferir paral·lelisme, sinó proporcionar una interfície amigable i coneguda al programador per a que ell indiqui quines funcions s'han d'executar en una unitat d'execució determinada. La llibreria està composta per tres operacions: `helf_execute`, `helf_exit` i `valid_arch`. Per entendre-les, presentem a la Taula 3 l'especificació d'aquestes funcions, amb els paràmetres que tenen, el seu retorn i una descripció del que realitzen.

Funció	Paràmetres	Retorn	Descripció
`helf_execute`	- `fun`: Funció a executar - `par`: Paràmetres de la funció - `ret`: Punter on es vol guardar el resultat de la funció - `arq`: Identificador de l'arquitectura on s'ha d'executar	- -1 si error - 0 altrament	Si `fun` o `arq` són nuls o no vàlids, retorna error i no fa res. Sinó, el procés o thread que la invoca saltarà a una unitat amb arquitectura `arq` a executar la funció `fun`. Per retornar, al final de la funció `fun` haurà d'invocar a `helf_exit` amb els paràmetres adequats.
`helf_exit`	- `ret`: Retorn de la funció - `arq`: Identificador de l'arquitectura a la qual es vol retornar	- -1 si error - 0 altrament	Si `arq` no és vàlid, retorna un error i no fa res. Sinó, retorna al punt just després d'on aquest procés ha invocat `helf_execute` per darrer cop.
`valid_arch`	- `arq`: Identificador d'arquitectura	- Cert o fals	Indica si l'identificador d'arquitectura passat com a paràmetre és vàlid (cert) o no (fals).

Taula 3: Descripció de les funcions de la llibreria HELF

A la Figura 14 es presenta el mateix exemple il·lustrat a la Figura 12, i adaptat per a que utilitzi la llibreria HELF per anar executant les diverses funcions que apareixien en aquest exemple, per acabar d'entendre les funcionalitats de la llibreria. En aquest exemple, el processador principal seria un x86 de 32 bits. Com es pot veure, el programador haurà d'adaptar els paràmetres d'entrada guardant-los en una estructura, i passant com a paràmetre un punter a aquesta estructura; i el retorn , ja que enlloc de retornar el valor s'haurà de retornar l'adreça. Aquest mecanisme és el mateix que es realitza amb la llibreria *pthread*, on els paràmetres d'entrada i sortida s'han d'adaptar convenientment.

```
int main(int argc, char *argv[]) {
    double res[6] = {0.0};
    helf_execute(&FFT, &paramsFFT, &res[1], ARCH_X64);
    helf_execute(&SOR, &paramsSOR, &res[2] , ARCH_X64);
    helf_execute(&MonteCarlo, &paramsMC, &res[3] , ARCH_SPE);
    helf_execute(&SparseMatMul, &paramsMM, &res[4] , ARCH_SPE);
    res[5] = LU (N, mintime, R);
    res[0]=(res[1]+res[2]+res[3]+res[4]+res[5])/5;
    print_out_results(&res);
    return 0;
}
```

```
#pragma x64
double FFT(struct FFT_params * p) {
     ...
     helf_exit(&resFFT, ARCH_X32);
}

#pragma x64
double SOR(struct SOR_params * p) {
     ...
     helf_exit(&resSOR, ARCH_X32);
}
```

```
#pragma spe
double MonteCarlo(struct MC_params * p) {
     ...
     helf_exit(&resMC, ARCH_X32);
}

#pragma spe
double SparseMatMul(struct SMM_params * p) {
     ...
     helf_exit(&resSMM, ARCH_X32);
}
```

```
double LU (int N, double min_time, Random R) {...}
```

Figura 14: Scimark2 separat en funcions per ser executades en unitats especialitzades i amb crides a la llibreria HELF

Amb aquest exemple, podem veure com el flux d'execució començaria al processador amb arquitectura x86 de 32 bits (sempre s'ha de començar en un processador de propòsit general), el thread aniria a executar la funció `FFT` a una arquitectura x86 de 64 bits (mitjançant la constant `ARCH_X64` que ho indica) i, a l'acabar l'execució, retornaria al processador principal. El mateix passaria amb la funció `SOR`. En el cas de les funcions `MonteCarlo` i `SparseMatMul` s'executarien en un accelerador SPE. Finalment, la funció `LU` s'executaria al processador principal al no estar marcada amb cap característica especial.

Tots aquests canvis d'unitat d'execució ens permetrien treure un major rendiment en l'execució d'aquesta aplicació, a diferència de si es fes tot al processador principal.

És necessari destacar que no es poden encadenar diversos canvis de processador sense fer el corresponent retorn al processador principal, sinó que cada `helf_execute` ha d'anar unit a un `helf_exit`. Entenem que aquesta restricció és assumible, ja que la idea d'aquests canvis de processador és per dur a terme una tasca molt específica en un accelerador especialitzat, que no requereix de cap altre nucli especialitzat. El motiu d'això es presentarà més endavant, tot i que si es necessités eliminar aquesta restricció només caldria revisar i adaptar el model.

A la Figura 15 es mostra un esquema d'aquestes crides encadenades no permeses.

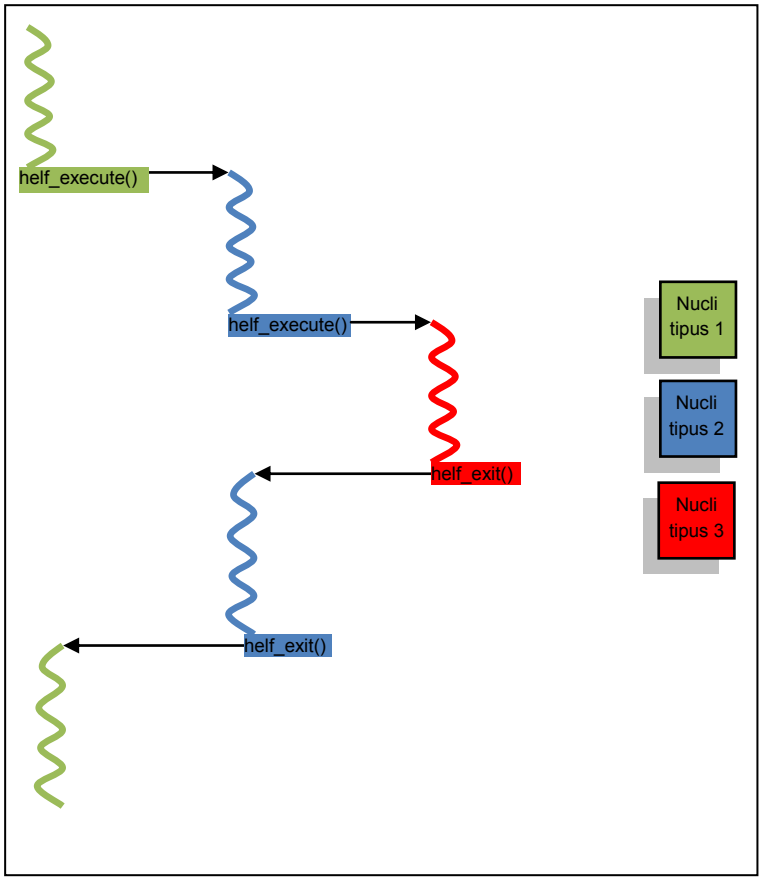


Figura 15: Crides encadenades, no permeses pel model

El que sí permet el model és un cas particular del que s'acaba de limitar, i és que cada processador o accelerador invoqui a un altre que no sigui el principal, però sense que el flux d'execució torni a passar per ell. És a dir, cada processador s'aniria passant la feina, i el darrer tornaria al processador principal. Per fer-ho, caldria adaptar lleugerament la llibreria, per indicar que estem en aquest cas particular.

Com es pot apreciar, en aquests casos un accelerador hauria de dur a terme un canvi de processador, tot i que hem dit que aquesta feina generalment la feia un processador de propòsit general. Per tant, si l'accelerador no és capaç de fer aquest canvi l'haurà de fer el processador de propòsit general. Aquest és un procediment molt habitual, donat que aquests acceleradors estan preparats per executar codi molt concret, però no per dur a terme tasques genèriques com seria el que proposem. De fet, i com es pot veure a l'ABI[10] del Cell [26], al capítol *Assisted SPE Library Calls*, quan el nucli SPE necessita fer una crida que no pot dur a terme, ha d'avisar al PPE mitjançant una senyal, establir una comunicació i enviar-se les dades entre ells. Per tant, caldria implementar aquesta sincronització de manera similar entre els processadors implicats.

A la Figura 16 es mostra un esquema d'aquest cas especial de canvi de context d'execució enllaçats que sí es permeten al nostre model.

[10] *Application Binary Interface*, interfície similar a una *API*, però enlloc de ser interfície a nivell de codi font ho és a nivell de binari. Per exemple, defineix com es passen els paràmetres d'una funció, com es fa el retorn, etc.

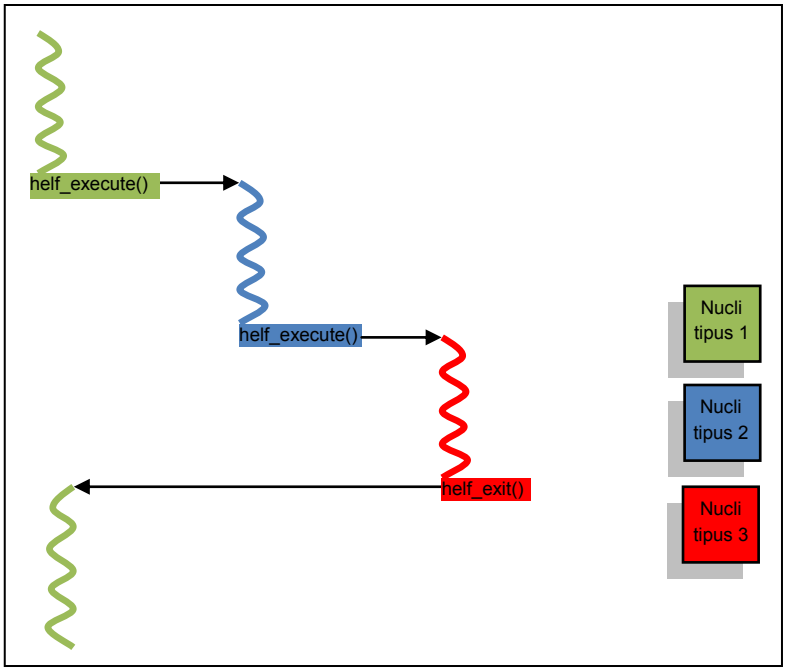


Figura 16: Cas particular de crides encadenades que sí són permeses pel model

Per que el programador pugui utilitzar les funcions de la llibreria, cal fer públiques les seves capçaleres i la resta d'informació necessària (com ara les constants). El programador, després, haurà d'incloure en els seus programes aquest fitxer perquè el procés de compilació no reporti cap avís. El fitxer, que està definit a `/usr/include`, s'anomena `libhelf.h` i conté les definicions que es presenten a la Figura 17.

```
#define ARCH_ARM        1
#define ARCH_FPGA       2
#define ARCH_PPC        4
#define ARCH_SPE        8
#define ARCH_USP        16
#define ARCH_X32        32
#define ARCH_X64        64

#define helf_arch(x) __attribute__((section (".text."#x)))

int helf_execute (void * func, void * args, void * result, int arch);
int helf_exit (void * result, int arch);
int valid_arch (int arch)
```

Figura 17: Fitxer de capçaleres i constants de la llibreria HELF

La definició de la macro `helf_arch(x)` és el que utilitzem per marcar les funcions,

és a dir, la manera com hem implementat el que està definit a la proposta mitjançant `#pragma`, i són atributs del compilador `gcc`.

L'estructura del binari no canviaria respecte el presentat a la Figura 13, donat que l'únic que hem fet és afegir crides a la llibreria, però les funcions segueixen estan separades en funció de les seves característiques. És per això que no el tornem a presentar de nou.

Per acabar amb aquesta part de la llibreria, hem pogut veure com el model que seguim és similar a les continuacions del microkernel Mach. Quan un codi ha de continuar la seva execució en un processador específic, l'única informació que necessita és on ha de continuar (una adreça) i quines dades d'entrada necessita (els paràmetres). Això és exactament el que fa el model de continuacions, com es pot veure a [27]. També hem vist que té similituds amb la llibreria *pthread*.

6.3. Novetats del sistema operatiu

Una vegada presentades les modificacions introduïdes a la part d'aplicació i a la llibreria HELF, comentem les novetats afegides al sistema operatiu.

Partint de les modificacions presentades al capítol anterior, començarem explicant els canvis dels objectes que manega el kernel per gestionar les seccions del format HELF, i posteriorment explicarem les modificacions realitzades per donar suport a les necessitats de la llibreria HELF, presentades a l'apartat anterior.

6.3.1. Objectes de kernel

Fins ara, la `task_struct` disposava de dos nous camps, un per indicar si el procés que representa aquesta `task_struct` és de tipus HELF, i una estructura de dades amb nodes, objectes que representen les noves seccions introduïdes amb tota la informació que necessitem. Les dues modificacions que s'han realitzat en aquest capítol, i que a continuació explicarem, són:

- S'ha canviat el concepte de node, i s'han afegit dos nous camps a l'estructura del

node que ens facilitaran la seva gestió

- S'ha afegit un nou camp a la `task_struct` per guardar nova informació

En primer lloc, ha canviat el concepte del node. Ara ja no representa una funció, com en la proposta del capítol anterior, sinó que el node representa una ISA, ja que ara s'agrupen totes les funcions compilades per una ISA en una sola secció. En aquest sentit, cada node segueix representant una secció del format HELF, però a l'haver canviat la filosofia de les seccions (com s'ha explicat a l'apartat 6.1), també ho ha fet la del node.

Seguint analitzant el node, i basant-nos en la idea de la gestió dels manegadors de dispositius a Linux [28], on es manté una estructura que disposa d'un seguit de punters a funcions que implementen les operacions per obrir, tancar, llegir, escriure, etc. un dispositiu, el node disposarà de dos punters a operacions que implementen el salt i el retorn cap a l'arquitectura que aquest representa.

Finalment, s'ha afegit un nou camp a la `task_struct`, que ens servirà per guardar la informació necessària per dur a terme els salts del thread entre unitats d'execució.

A la Figura 18 es presenta un esquema d'un procés HELF amb aquestes dues novetats afegides en color gris. Com es pot veure, el node disposa de dos nous camps (per similitud amb els dispositius de Linux es diuen `open` i `close`), i la `task_struct` disposa d'un nou camp. El node que es mostra representa la ISA x86 de 32 bits, com es pot veure tant en el camp `section_type` com en els noms de les operacions per saltar entre unitats d'execució.

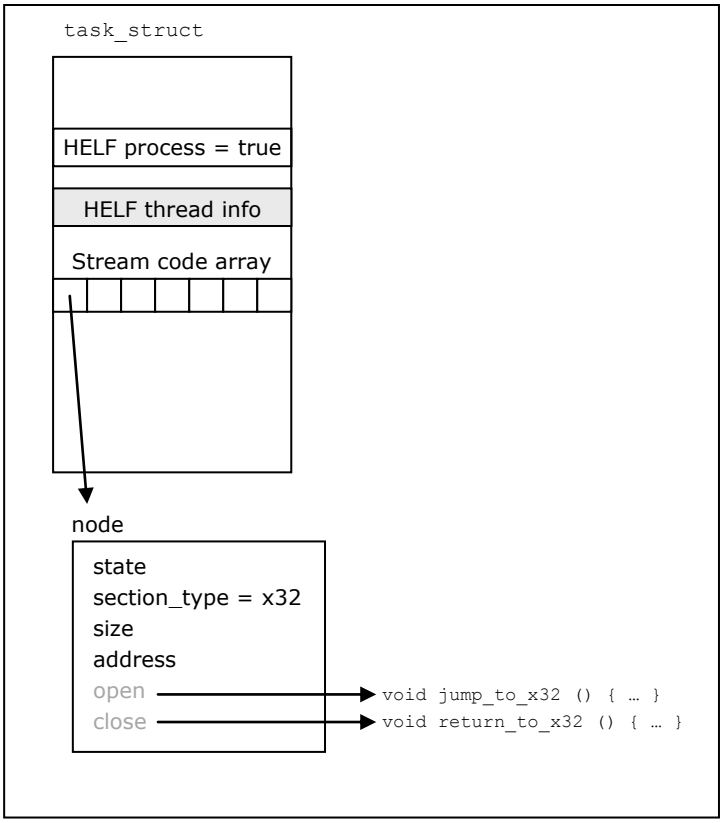


Figura 18: Estructura que representa un procés HELF amb les noves modificacions a nivell de kernel

Els punters del node els inicialitzarà, com tota la resta dels camps del node, el carregador específic del format HELF. Basant-se en el nom de cada secció, utilitzant el conveni establert a l'apartat 6.1, es detectarà quina ISA representa aquella secció, i farà que els camps `open` i `close` apuntin a les funcions corresponents. Aquestes funcions seran específiques de cada arquitectura, i caldrà crear tants parells de funcions `open-close` com arquitectures suporti el nostre sistema. El camp `section_type` ens identificarà quina ISA representa el node.

L'altre nou camp (`HELF thread info`) ens permetrà guardar l'estat del thread (el context d'execució, amb els registres del processador) abans de fer el salt a la nova unitat d'execució.

6.3.2. Suport a la llibreria HELF: crides a sistema

Com ja s'ha explicat a l'apartat 6.2, la llibreria HELF implementa dos mètodes principals: un

per transferir el control d'un thread a una unitat d'execució i l'altra per retornar aquest control a la unitat principal. Aquests dos mètodes necessiten suport del sistema operatiu, que tot seguit expliquem.

Els dos mètodes de la llibreria invoquen dues noves crides a sistema, que tenen els mateixos paràmetres que les respectives crides a sistema. Les capçaleres d'aquestes crides a sistema són les següents:

```
int start_hexec (void * fn, void __user * arg, void __user * ret, int arch)

int finish_hexec (void __user * ret, int arch)
```

Com ja es pot intuir, la crida a llibreria `helf_execute` invoca a la crida a sistema `start_hexec`, mentre que la crida a llibreria `helf_exit` invoca a la crida a sistema `finish_hexec`. Com amb el format HELF, la lletra "h" significa *heterogeneous*. Les dues crides a sistema tenen un comportament gairebé idèntic, i molt intuïtiu. Els passos que segueixen ambdues crides a sistema són els següents:

1. Comproven si el procés que executa la crida és de tipus HELF. Si no és així, retornen un error.

2. Obtenen el node que representa la ISA de l'arquitectura on es vol executar la funció, mitjançant una nova funció `find_node_by_arch (int arch)`. Aquesta funció retorna l'adreça del node que representa l'arquitectura `arch`, o `NULL` si no existeix tal node (si existeix, és únic pel que hem comentat anteriorment). Si no existeix retornarem un error, donat que no podem dur a terme el salt sense la informació associada al node.

3. En cas d'existir el node, invoquen el seu mètode `open` o `close`, depenent de quina crida a sistema sigui (`start_hexec` invoca el mètode `open`, i `finish_hexec` el `close`), amb tots els paràmetres menys el que codifica l'arquitectura (ja l'hem utilitzar per cercar el node que la representa). Per tant, s'invoca la crida específica

de l'arquitectura que representa el node, ja sigui la de saltar o retornar d'una unitat d'execució.

4. Al finalitzar el mètode específic per cada arquitectura, retornen un 0 indicant que tot s'ha realitzat correctament.

D'aquesta manera, les crides a sistema són molt flexibles, ja que introduint un nou node podrem invocar els seus mètodes sense haver de canviar el seu comportament.

6.3.3. Funcions específiques per canviar d'arquitectura

A continuació parlarem de les funcions específiques per dur a terme un canvi d'execució entre processadors, que seran apuntades pels camps `open` i `close` del node. La situació habitual serà la següent: una aplicació s'estarà executant en un processador de propòsit general, i el programador indicarà que vol canviar el context d'execució, enviant una tasca a un altre processador (un accelerador, un altre processador de propòsit general, etc.). La pregunta és: quins aspectes hem de tenir en compte per dur a terme això?

Com ja hem estudiat el funcionament de la llibreria *pthread* coneixem certs aspectes que cal tenir en compte. La llibreria *pthread* crea un fil d'execució diferent al pare, i no hi ha cap canvi de nucli (a no ser que es planifiquin en paral·lel en un multiprocessador). Per tant, fixant-nos en aquest model podem establir un conjunt de coses a tenir en compte en un entorn homogeni:

- Adreça de l'espai d'usuari on s'ha de continuar l'execució. La llibreria *pthread* fa (en arquitectura x86) un `call`, que modifica el registre `%eip`, cosa que podem fer des del sistema operatiu. Com tot el codi del programa ja està carregat a memòria no cal preocupar-se de fer-ho.
- Paràmetres a la pila d'usuari. Tot i que en el cas dels *pthreads* el thread disposa d'una nova pila d'usuari, diferent a la del pare, en el nostre cas no és necessari, ja que podem utilitzar la mateixa pila (donat que volem que sigui el mateix thread el que realitzi l'execució). L'únic que caldrà fer, per tant, és apilar a l'espai d'usuari l'adreça de l'estructura on s'han guardat els paràmetres. Com en el cas dels

pthreads, al compartir memòria això no comporta cap problema, ja que podrem accedir a aquests paràmetres.

- Retorn. Si volem accedir a memòria que pertany al pare, com la memòria és compartida entre tots els threads no cal preocupar-se d'aquest fet.

En canvi, l'objectiu d'aquest Projecte Final de Carrera és realitzar una proposta, i no només hem de pensar en un entorn homogeni, sinó també en un d'heterogeni. Si pensem en un entorn heterogeni, ens sorgirien més preguntes:

1. *Què passa si no està carregat el codi o les dades que ha d'executar el thread, i/o l'accelerador utilitza memòria pròpia no compartida?*

En aquest cas, el sistema operatiu és l'encarregat de carregar el codi o les dades allà on l'accelerador els necessiti o, si el que es demana és l'acció contrària com en el cas del retorn de resultats, copiar les dades de l'accelerador a la memòria del processador principal. Aquesta tasca, que a priori no té més complicació, és costosa a nivell d'implementació, donat que cal anar dient quants bytes s'han de copiar d'una banda a l'altra, a partir de quina adreça s'han de copiar, on s'han de guardar, etc.

2. *Com es representa la nova tasca al processador on es vol enviar la feina?*

Com no estem modificant el model d'execució, farem el que estableixi el model d'execució ja existent en cada cas. En el cas d'un processador de propòsit general, un procés o un thread es representen ambdós mitjançant una `task_struct` inserida en una cua (`run_queue`) on es representen totes les tasques que estan llestes per ésser executades, i quan es decideix fer un canvi de context, el planificador de tasques tria de la cua quina s'executa en funció de la política que segueixi.

En el cas que canviéssim l'execució a un processador que funcioni de manera similar (per exemple, podríem sol·licitar el canvi d'un processador x86 a un PowerPC), caldria:

- Bloquejar la `task_struct` del processador principal, traient-la de la `run_queue` perquè no es planifiqués (ja que l'execució es durà a terme a un altre nucli, i és absurd mantenir la tasca gastant CPU al processador origen) i triant un altre procés per ocupar la CPU mitjançant un canvi de context.

- Adaptar aquesta tasca (si és necessari) o crear-ne una de nova perquè representi una tasca del processador destí. A aquesta nova tasca caldrà associar-li tota la informació que sigui necessària (copiar el codi, les dades, etc. a la memòria que té associada la tasca), inserir-la a la cua de tasques que executa el processador destí, i esperar que el planificador específic la triï per ésser executada.
- A l'acabar l'execució, copiar el retorn del processador destí a la `task_struct` original, tornar a inserir-la perquè es planifiqui i destruir la tasca del processador destí.

D'aquesta manera, tindrem dues estructures que representen al mateix thread, una a cada processador, tot i que una serà la "imatge" de l'altra, adaptada a les necessitats de cada planificador.

En cas que el processador destí no disposi de planificador, caldria veure si està lliure o ocupat. Si està ocupat, caldria esperar a que s'alliberés. Quan estigui lliure, caldria seguir els passos necessaris per dur a terme una execució (carregar el codi, les dades, etc.) i ordenar l'inici de l'execució. A l'acabar retornaria el control de l'execució al processador principal.

3. *Com s'atura un procés que està en un altre processador? Com se li envia una senyal (signal)?*

En aquests casos, de la mateixa manera que es disposen d'operacions per canviar el context d'execució entre processadors, caldria tenir operacions específiques per cadascuna d'aquestes funcions. Es podrien tenir operacions per copiar resultats, tractar *signals*, guardar i restaurar l'estat de l'execució, etc. i que el node inclogués totes les accions que fossin dependents de l'arquitectura, no només les dues que hem presentat. Totes aquestes funcions serien específiques per cada arquitectura, i no es pot generalitzar el seu comportament, tot i que hauríem de seguir el model que utilitzen habitualment.

Com es pot veure, tots aquests aspectes no són trivials i caldria tenir-los en compte si s'implementés aquesta proposta a aquest nivell. Nosaltres, en canvi, hem implementat la proposta en un sistema amb arquitectura multiprocessador homogènia x86, donat que el

que s'ha explicat és d'una magnitud superior al que es pot realitzar en un Projecte Final de Carrera. Com ja hem dit, la proposta té en compte molts més aspectes dels que s'han acabat implementant.

Els passos que realitza la funció específica de l'arquitectura x86 per transferir el control d'un thread són els següents:

1. Obté els registres de la pila de kernel del procés, que s'hauran guardat automàticament a l'entrar al sistema operatiu. Aquesta informació representa el context d'execució.

2. Els guarda al nou camp de la `task_struct` (`HELF thread info`), juntament amb l'adreça on s'ha de guardar el resultat de la funció a executar. Per aquest motiu, el model no permet fer salts indefinits entre processadors sense retornar (Figura 15), ja que no només hauríem de guardar un context sinó un nombre indefinit de contextos, un per cada salt. Per simplificar el model hem pres aquesta decisió, tot i que amb una pila es podria implementar fàcilment. En el cas de que es facin diversos salts però es retorni al processador que ha realitzat el primer canvi de processador (Figura 16) només guardarem el context d'execució del primer salt.

3. Canvia el registre `%eip`, fent que valgui el valor de la funció que volem executar.

4. Apila els paràmetres que necessita la funció a la pila d'usuari del procés.

Com tot el codi del programa ja està carregat a memòria, al sortir del sistema operatiu el thread passarà a executar la funció indicada, i es trobarà els paràmetres a la pila tal i com els aniria a buscar en una execució "estàndard". Aquesta funció, al finalitzar, haurà de retornar el control al processador principal, amb la crida a la llibreria HELF afegida pel programador, que en el cas de l'arquitectura x86 realitza els passos següents:

1. Restaura els registres, guardats abans d'executar la funció.

2. Copia el resultat de la funció a l'adreça que ens vam guardar abans de canviar de processador.

Al sortir del sistema operatiu i tornar a usuari, haurem restaurat l'estat d'execució del procés tal i com estava abans d'haver canviat de processador. Per tant, seguirem executant la instrucció posterior al canvi de processador, amb tots els registres, pila, etc. en el mateix estat. També tindrem el resultat de l'execució a on s'havia indicat. En resum, totes les accions que calia fer s'hauran realitzat correctament.

A la Figura 19 es mostra un resum del flux d'execució seqüencial d'una aplicació utilitzant la llibreria, i els passos que fa el sistema operatiu, el canvi de processador i el retorn al processador principal. La imatge s'ha fet genèrica, com si intervinguessin dos processadors diferents, i caldria tenir en compte tots els aspectes que s'han estudiat a la proposta però no s'han implementat. Aquesta imatge intenta resumir les tasques que cal fer internament, i com es reflectiria en el flux d'execució d'una aplicació.

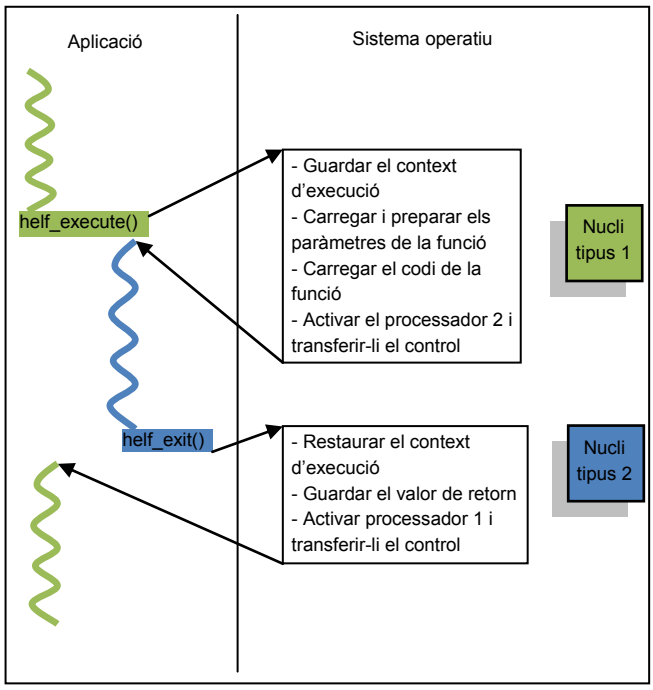


Figura 19: Resum del flux d'una aplicació amb la llibreria i la visió del sistema operatiu

Amb les solucions presentades en aquest apartat, la llibreria sempre executa la mateixa crida a sistema, sigui quina sigui l'arquitectura (i és en funció de l'arquitectura que

es crida a una operació específica o a una altra), fent que la solució sigui senzilla pel programador i la llibreria. Així tenim en una sola entrada totes les funcions que gestionen l'execució i el retorn dels diferents processadors que implementem, de manera similar al que passa a Linux amb les crides a sistema.

Igualment, la nostra proposta és similar a la política de Linux envers els dispositius, on hi ha una part que és independent del dispositiu (la API és comuna, independentment del dispositiu al qual es faci la petició) i part que és dependent. En el nostre cas, el punt d'entrada i de sortida són independents de la unitat d'execució a la que es vol transferir el control, i la part dependent són les diferents funcions que implementen el servei per cada arquitectura.

Com hem dit anteriorment, la nostra proposta també té similituds amb la gestió dels manegadors de dispositius de Linux, amb una estructura que conté punters als mètodes propis d'aquella estructura.

Per tant, creiem que la decisió final presa és satisfactòria, i que el disseny realitzat és robust, fàcilment escalable i entenedor, ja que hem utilitzat idees de disseny ja realitzats en sistemes molt usats i estudiats.

6.4. Proves i resultats

En aquest apartat es presenten les proves que s'han realitzat en aquesta segona etapa i els resultats que s'han obtingut. A diferència de les mesures fetes a la primera etapa (on només s'havia avaluat el rendiment i la sobrecàrrega afegida al sistema operatiu), en aquesta segona etapa mesurarem el rendiment a nivell d'aplicació, ja que el que més s'ha perfeccionat en aquesta etapa ha estat el model d'execució, el punt clau perquè les aplicacions puguin aprofitar l'heterogeneïtat del *hardware* a l'hora d'executar-se. També volem provar que tot el que s'ha realitzat, començant per l'aplicació i acabant pel sistema operatiu, s'ha desenvolupat correctament.

Les primeres proves que s'han fet han estat per avaluar la correctesa i la robustesa

del model desenvolupat, pensant casos buscats a propòsit per intentar trobar algun error. És a dir, són uns tests artificials que ens serviran per veure fins a quin punt es controlen o no els errors que se'ns acudeixin.

A la segona part de les proves analitzarem el comportament d'una aplicació ja presentada, el *benchmark* Scimark2, en una arquitectura multiprocessador homogènia.

6.4.1. Avaluació de la correctesa i robustesa del model

Podem diferenciar tres grans blocs a l'hora de provar el nostre model. Primerament, la part corresponent a l'aplicació, on intervé el procés de compilació i enllaçat. En segon lloc, la part de la llibreria HELF, que rep les peticions de les aplicacions per canviar el flux d'execució entre els diferents processadors. Finalment, la part del sistema operatiu és l'encarregada de realitzar els canvis de processador i de proporcionar la informació que la llibreria necessita. Per tant, les proves que es faran a continuació aniran enfocades a provar aquests tres aspectes.

La part que menys cal provar és la de l'aplicació, donat que la generació del binari és (o hauria de ser) força automàtica, interpretant la informació afegida a nivell de funció.

A la Taula 4 es mostren les proves realitzades, juntament amb el responsable de comprovar el cas erroni, i la resposta que ha tingut el sistema.

Situació	Responsable	Reacció
Codi font que marqui una funció amb una arquitectura no suportada pel format HELF, o que el nom que es proporciona no sigui vàlid.	Enllaçador	Genera el binari ignorant aquesta informació, incloent aquesta funció dins la secció de codi principal (`.text`).
Invocació a `helf_execute` o `helf_exit` amb paràmetres nuls.	Llibreria	Si el punter a la funció, en el cas de `helf_execute`, és nul, o l'arquitectura, en el cas de les dues funcions, no és vàlida, retorna un error. La resta de paràmetres poden ser nuls, ja que una funció pot no necessitar paràmetres ni retornar un resultat.

Invocació a `helf_execute` amb un punter a funció invàlid.	No definit	És impossible verificar que el punter conté realment l'adreça d'una funció. Si l'adreça on apunta no pertany a l'espai de memòria del procés, serà detectat pel sistema operatiu i es produirà una excepció de violació de segment. Sinó, el comportament que tindrà serà inesperat i impredictible, i en cap cas el correcte.
Execució d'un binari ELF, que invoca funcions de la llibreria HELF.	Sistema operatiu	Detecta que el procés que representa l'aplicació no és un procés HELF i retorna error.
Execució d'un binari HELF sense cap secció heterogènia, que invoqui `helf_execute` per a fer algun salt entre processadors.	Sistema operatiu	El sistema no disposarà de la informació per aquella arquitectura i no podrà dur a terme el salt. Retorna un error.
Execució d'un binari HELF, on es fa una crida a `helf_exit` abans de la seva corresponent a `helf_execute`.	Sistema operatiu	Aquest aspecte no es controla, i no podem saber el comportament que tindrà el sistema. El més segur és que l'aplicació acabi de manera inesperada.

Taula 4: Proves realitzades per avaluar la correctesa del model amb la resposta obtinguda

Com es pot veure, els casos d'error estan controlats i delimitats, i són casos que farien que l'aplicació deixés de funcionar. Pel que hem pogut veure, principalment hi ha dos casos crítics.

En el cas de proporcionar una adreça de funció invàlida, el procediment habitual és mitjançant el nom de la funció, i és el compilador qui tradueix aquest nom a l'adreça de memòria on comença aquesta funció. Per tant, és un error que difícilment es pot donar, a no ser que es faci intencionadament. Aquest error també el tindria la llibreria de *pthreads*, ja que només controla que el paràmetre no sigui nul.

Per altra banda, determinar que es realitza un `helf_exit` abans que un `helf_execute` és senzill de controlar (mitjançant un *flag* a dins del sistema operatiu que ho indiqui) i caldria tenir-ho en compte per fer més robust el sistema.

6.4.2. Scimark2

Una vegada hem trobat els punts febles del sistema implementat fins ara, anem a

comprovar que una aplicació sense cap error que pugui desestabilitzar el sistema es comporta correctament. L'aplicació que utilitzarem com a prova és el *benchmark* Scimark2, ja presentat anteriorment. L'Scimark2 conté cinc nuclis computacionals que realitzen càlcul numèric i científic, i està disponible tant en llenguatge Java com en C. Al finalitzar l'execució reporta a l'usuari una puntuació on aproxima el rendiment obtingut en l'execució (en *MFlops*, milions d'instruccions en coma flotant per segon). Aquest serà, per tant, un dels paràmetres que compararem.

A part d'aquesta mesura que ja ens ofereix la pròpia aplicació, calcularem el temps d'execució, que ens servirà com a segon paràmetre a comparar.

Realitzarem dos tipus de proves: execució seqüencial i paral·lela (amb *pthreads*) dels cinc nuclis computacionals que componen el *benchmark*. Aquestes dues proves es faran amb la versió original del benchmark i amb la versió modificada utilitzant la infraestructura explicada. A continuació es presenten les proves, realitzades en un ordinador amb un processador Intel® Core2Duo a una freqüència de 1,86 GHz i 2GB de memòria RAM. Com en les proves presentades a l'anterior capítol, l'entorn d'execució ha estat controlat, aïllant al màxim l'execució del *benchmark*, de tal manera que tots els recursos de la màquina estiguessin disponibles. Com en les proves del capítol anterior, s'han realitzat tantes execucions com ha estat necessari per apreciar que els resultats eren prou estables i el seu resultat no variava excessivament (unes cinc execucions).

Com el lector ja pot intuir, en aquestes proves no hi ha cap salt a un processador especialitzat en tasques numèriques, donat que és un processador homogeni. Tot es realitzarà, doncs, al mateix processador, encara que s'utilitzarà la infraestructura construïda per fer els salts. Per això, els resultats que s'obtindran entre les execucions originals i les que utilitzin el nostre sistema seran molt semblants.

El que sí permetran analitzar aquestes proves és si el nostre sistema penalitza molt l'execució, i si ens ofereixen la possibilitat de saltar a un processador especialitzat si aquest existís. Com ja es va introduir a l'apartat dels objectius d'aquest Projecte, no es busca tant l'aconseguir un gran rendiment, sinó el proporcionar una interfície flexible per a que

aquests salts es puguin dur a terme. Per tant, aquestes proves sí que ens serveixen per avaluar els objectis que ens hem fixat al començar el Projecte.

Rendiment d'una execució seqüencial, comparant les versions original i HELF

En aquesta primera prova comparem el rendiment dels cinc nuclis computacionals entre l'aplicació original i la versió HELF fent una execució seqüencial. A la Taula 5 es presenten els resultats d'aquesta prova.

Nucli computacional	Original	Versió HELF
FFT	138,87	138,43
SOR	474,62	465,45
MonteCarlo	46,60	46,74
Sparse matmult	190,18	190,62
LU	249,52	250,98

Taula 5: Comparativa de rendiment d'una execució seqüencial del *benchmark* Scimark2

El rendiment entre les dues versions és similar, ja que en alguns casos és més elevat el de la versió original (com en el cas del `SOR`) mentre que en altres ho és més la versió HELF. Però en general el rendiment obtingut és similar, cosa que indica que la nostra proposta no penalitza el rendiment de l'aplicació.

Temps d'una execució seqüencial, comparant les versions original i HELF

A la Figura 20 es presenten els temps totals, en segons, de l'execució seqüencial comparant la versió original i la versió HELF.

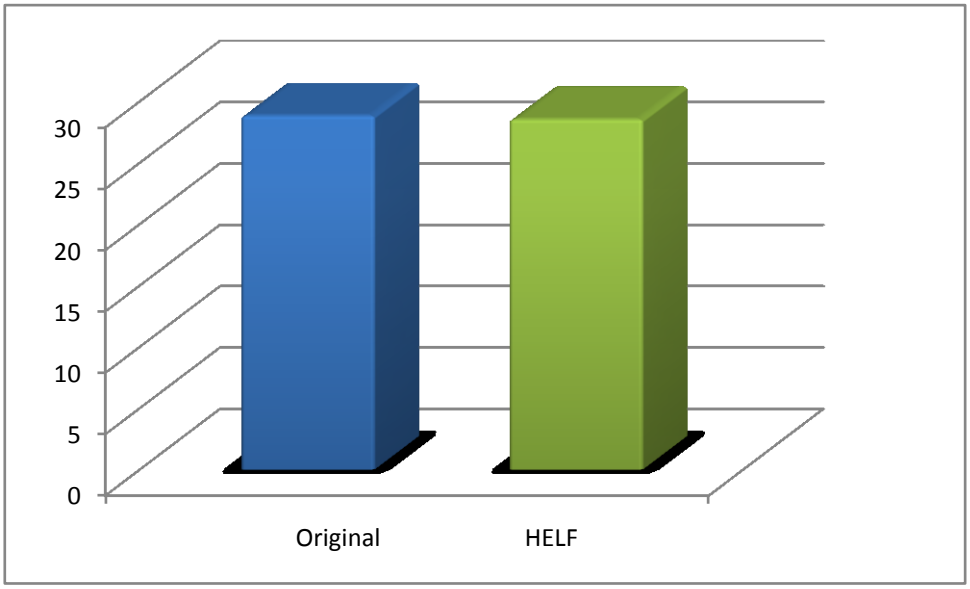


Figura 20: Comparativa de temps d'una execució seqüencial del *benchmark* Scimark2

En aquest cas, tarda lleugerament menys la versió utilitzant la nostra proposta que la versió original, tot i que la diferència és mínima.

Rendiment d'una execució paral·lela, comparant les versions original i HELF

En aquesta segona prova comparem el rendiment dels cinc nuclis computacionals entre l'aplicació original i la versió HELF, però fent que cada nucli l'executi un *pthread* independent. A la Taula 6 es presenten els resultats.

Nucli computacional	Original	Versió HELF
FFT	38,08	30,36
SOR	99,09	108,86
MonteCarlo	12,12	14,12
Sparse matmult	37,83	56,99
LU	73,01	68,44

Taula 6: Comparativa de rendiment d'una execució paral·lela del *benchmark* Scimark2

Com en el cas de l'execució seqüencial, el rendiment entre les dues versions és similar, tot i que en aquest cas les diferències són majors, i rendeix millor la versió adaptada que no pas l'original. Tot i això, hi ha certs casos en que treu major rendiment la versió original, com també succeïa abans.

Temps d'una execució paral·lela, comparant les versions original i HELF

Finalment, a la Figura 21 es presenten els temps totals, en segons, de l'execució paral·lela, comparant la versió original i la versió HELF.

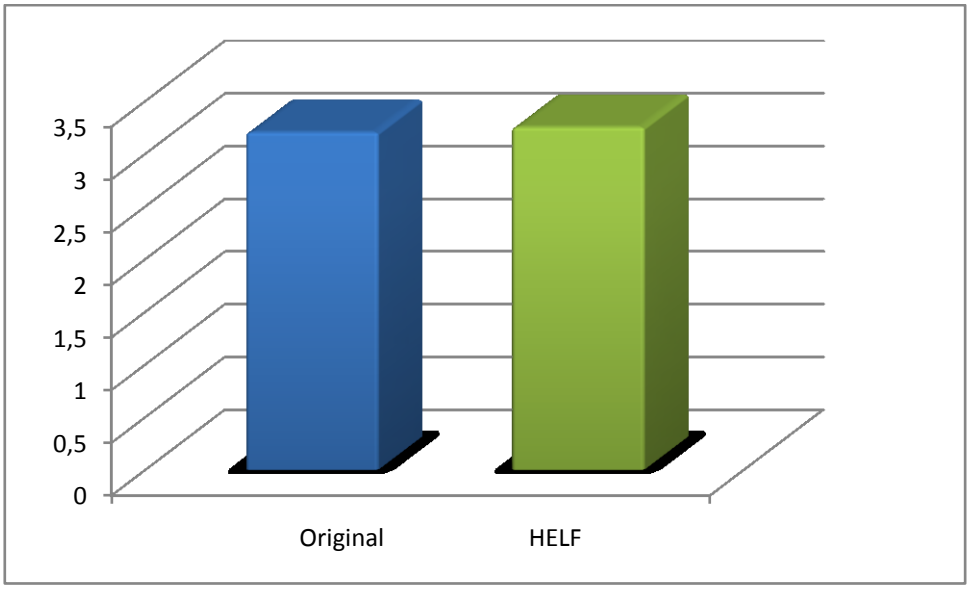


Figura 21: Comparativa de temps d'una execució paral·lela del *benchmark* Scimark2

En aquest cas, el que crida l'atenció es la diferència de temps respecte l'execució seqüencial. Mentre que aquesta tardava uns 25 segons, utilitzant *pthreads* l'aplicació tarda uns 3 segons. Per tant, paral·lelitzar l'aplicació ha servit per disminuir molt el seu temps d'execució, unes 8 vegades més ràpid (*speed-up* de 8X). Comparant entre sí la versió original i la versió adaptada, veiem que els temps d'execució són gairebé idèntics.

6.5. Conclusions

En aquest darrer apartat presentem les conclusions a les que hem arribat després d'implementar totalment la proposta per una arquitectura multiprocessador homogènia, i haver tingut en compte aspectes per la implementació en un entorn heterogeni més genèric. Podríem dir que el que s'ha fet és abordar les limitacions aparegudes en la primera etapa i afrontar aspectes que havien quedat en un segon pla.

En primer lloc, s'ha perfeccionat l'extensió HELF, fent que el codi no es separi a nivell de funció, sinó a nivell d'ISA. Per tant, l'inconvenient que l'extensió tenia que hi hagués moltes noves seccions ha desaparegut ja que ara n'hi haurà, com a màxim, el

nombre d'arquitectures que el sistema suporti. Això també ha implicat que l'extensió HELF tingui una classificació més lògica i clara, donat que no hi ha cap motiu perquè les funcions estiguin separades en seccions, a diferència que estiguin separades en funció de la ISA per la qual estan compilades.

En segon lloc, hem tractat el model d'execució, basant-nos en una llibreria que permet al programador indicar quina funció vol executar, amb quins paràmetres, on s'ha de deixar el seu retorn i l'arquitectura on vol que s'executi. Amb similituds amb la llibreria *pthreads*, i amb capacitat perquè el programador pugui introduir paral·lelisme, aquesta llibreria només s'ocuparà d'oferir gestió per l'heterogeneïtat de la màquina. No volem, per tant, que el programador es veiés obligat a usar el paral·lelisme que nosaltres implementéssim, sinó el que decideixi. A més, qualsevol salt a una unitat d'execució segueix el mateix camí, fent que el sistema sigui molt flexible a l'hora d'afegir noves funcionalitats i suport per noves arquitectures.

En tercer lloc, el sistema operatiu s'ha adaptat per gestionar, per una banda, les novetats introduïdes a l'extensió HELF, i per altra banda les millores de la llibreria. S'ha proposat una gestió més intuïtiva dels objectes de kernel que representen les noves seccions, tenint punters a les operacions que fan el salt entre unitats d'execució, seguint la política dels manegadors de dispositius a Linux. S'han abordat aspectes que caldria tenir en compte en arquitectures heterogènies, i s'han implementat les operacions específiques per saltar i tornar l'execució d'un thread en una arquitectura x86.

Finalment, hem fet mesures de la proposta. El primer conjunt de proves ha estat dissenyat per avaluar la correctesa i robustesa del sistema amb exemples erronis buscats expressament, per veure si el sistema ho detectava i responia correctament. Pel segon conjunt de proves hem utilitzat el *benchmark* Scimark2, comparant el rendiment i temps d'execució entre la versió original i la versió adaptada per utilitzar el nostre model d'execució. S'han fet execucions seqüencials i utilitzant threads.

Totes les proves han estat satisfactòries, no tant obtenint un major rendiment com oferint al programador la possibilitat de canviar d'unitat d'execució d'una manera intuïtiva i còmoda, objectius plantejats al començar el desenvolupament del Projecte.

7. MIGRACIÓ A UNA ARQUITECTURA HETEROGÈNIA: CELL BEA

En aquest darrer capítol dels tres dedicats al desenvolupament del Projecte parlarem de com s'ha migrat la proposta presentada fins ara per aprofitar un entorn heterogeni. Tot i que el disseny de la proposta s'ha fet pensant en un entorn heterogeni, i podríem estar satisfets per tot el que s'ha fet sense fer una migració real, volíem intentar fer aquesta migració per veure realment fins a quin punt el nostre sistema seria útil i fàcilment adaptable a un nou entorn heterogeni força desconegut, com és ara per ara l'arquitectura del Cell per nosaltres.

Molta part d'aquesta migració ha calgut dedicar-la, doncs, a l'estudi d'aquesta nova arquitectura. Com aquest és un Projecte Final de Carrera enfocat a temes de recerca, també és un dels objectius que es perseguien, i només per això l'haver fet la migració a una arquitectura heterogènia és una experiència molt enriquidora.

Havent acabat, per tant, la implementació en un sistema multiprocessador heterogeni, i com encara teníem al voltant d'un mes per finalitzar el Projecte, vam decidir migrar la proposta cap a una arquitectura heterogènia, on adquireix un sentit molt més ple. L'arquitectura heterogènia utilitzada ha sigut la del Cell BE, donat que al departament disposem de dues PlayStation 3, el nucli de les quals és un Cell BE. D'aquesta manera, el Projecte esdevé totalment complet, perquè provarem la proposta en un entorn heterogeni.

Per tant, l'objectiu en aquesta darrera etapa és un de sol, i molt clar: migrar tota la proposta al Cell BE, amb una arquitectura multiprocessador heterogènia.

Com ja s'ha explicat breument a l'apartat 3.1.1, el Cell BE és un processador que disposa d'un nucli de propòsit general (arquitectura PowerPC) i vuit nuclis especialitzats en operacions vectorials i multimèdia. Caldrà, doncs, adaptar tot el que s'ha explicat als darrers dos capítols per treure profit d'aquest entorn.

Per afrontar aquest objectiu, farem quelcom similar al que hem fet durant tot el Projecte: dividir la feina en graus de dificultat per afrontar-la amb més garanties. De la mateixa manera que abans de fer el salt a una arquitectura heterogènia hem implementat el sistema en un entorn homogeni, en aquesta etapa també dividirem en dos etapes

aquesta migració:

- Migrar la proposta a l'arquitectura PowerPC, sense utilitzar els SPE's
- Finalitzar la migració incorporant la part dels SPE's

El capítol està estructurat de la següent manera: els dos primers apartats d'aquest capítol seran una introducció al Cell BE i a l'arquitectura PowerPC, respectivament, per situar l'entorn de treball al que ens hem d'adaptar. Així coneixerem les seves peculiaritats i diferències respecte una arquitectura x86.

Al tercer apartat adaptarem el model vist fins ara a l'arquitectura PowerPC, sense tenir en compte els vuit acceleradors de què disposa el Cell BE. Per tant, implementarem la proposta per l'arquitectura PowerPC únicament (que serà la segona arquitectura en què s'haurà provat el nostre sistema, juntament amb el que ja funciona en l'arquitectura x86), deixant de banda l'existència dels processadors SPE. És a dir, no contemplarem la heterogeneïtat del Cell. El canvi d'arquitectura ja és un fet prou important com per dedicar-li un sol apartat. Un cop acabat això es podria avaluar la proposta (com s'ha fet amb l'arquitectura x86), però s'ha decidit fer les proves quan s'hagi implementat tot el conjunt.

El quart apartat d'aquest capítol serà el que presentarà la proposta final del Projecte Final de Carrera, amb la utilització dels coprocessadors SPE. En aquest punt és quan podrem dir que el sistema ja ha estat provat en una arquitectura heterogènia, assolint el principal objectiu i motiu d'aquest Projecte Final de Carrera.

Posteriorment, realitzarem un conjunt de proves per analitzar tota la proposta, i extraurem unes conclusions del treball realitzat.

7.1. Cell Broadband Engine

En aquest apartat s'introdueix el processador Cell, donat que és amb el que es treballarà en aquest capítol. El processador Cell és el sistema multiprocessador heterogeni que hem triat per desenvolupar aquesta darrera part del Projecte. Segurament, altres sistemes

heterogenis també ens servirien (combinació d'un processador x86 i un PowerPC, o un processador de propòsit general amb una o més GPU's, etc.), però utilitzem el Cell perquè el tenim disponible i podem fer la migració real.

Com ja s'ha comentat a l'apartat 3.1.1, el Cell BE disposa d'un nucli PowerPC de propòsit general anomenat *Power Processing Element* (*PPE*) i vuit coprocessadors anomenats *Synergistic Processing Element* (*SPE*) especialitzats en operacions vectorials i multimèdia a gran velocitat. Els processadors estan connectats per un bus intern amb un gran ample de banda anomenat *Element Interconnect Bus* (*EIB*). A la Figura 22 es pot veure un esquema del Cell, on es distingeixen aquestes parts. Tot seguit s'expliquen amb més detall els processadors (tant el de propòsit general com els específics), ja que els detalls arquitectònics d'ambdós són els dos aspectes més importants pel desenvolupament de la nostra proposta.

7.1.1. Power Processing Element (PPE)

Aquest processador, de la família dels PowerPC, és l'encarregat d'executar el nucli del sistema operatiu i de controlar l'execució dels SPE's. És un processador que permet executar dos fils d'execució simultàniament, de tal manera que pel sistema operatiu és com si hi hagués dos processadors virtuals. Una aplicació pel Cell no hauria d'utilitzar excessivament aquest processador, donat que l'objectiu d'aquest nucli no és aconseguir un gran rendiment per sí sol. El que farà aquest processador, doncs, és repartir les tasques a fer, distribuint-les entre els coprocessadors perquè siguin aquests qui facin l'execució. Quan aquests acabin, serà també l'encarregat de recollir els resultats obtinguts.

7.1.2. Synergistic Processing Element (SPE)

L'SPE és un coprocessador especialitzat en tasques vectorials i multimèdia, que té algunes peculiaritats. En primer lloc, disposa d'una memòria local (on va a buscar tota la informació), independent de la memòria cache i RAM del xip. Aquesta memòria és molt ràpida, i és on es guarda el codi del programa a executar, juntament amb les dades que li han estat assignades per processar.

En segon lloc, l'SPE disposa de 128 registres de 128 bits de propòsit general (*GPR*). Aquests registres permeten executar operacions SIMD (executar la mateixa instrucció sobre un conjunt gran de dades a la vegada, com ara sumar dos vectors) en blocs de dades de diferents mides:

- 16 enters codificats amb 8 bits
- 8 enters codificats amb 16 bits
- 4 enters codificats amb 32 bits
- 4 nombres reals en coma flotant

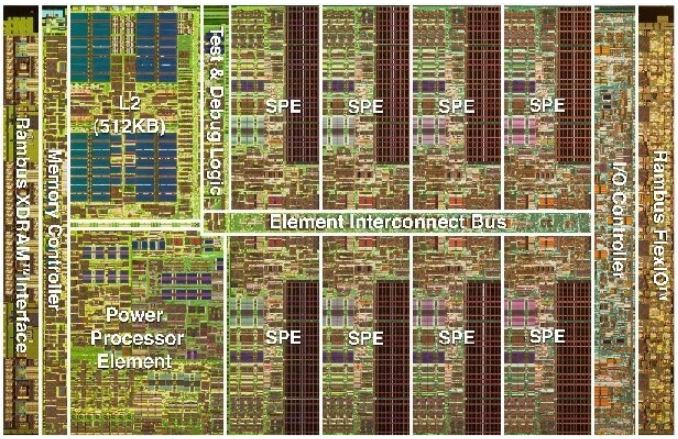


Figura 22: Esquema de les diferents parts del Cell

7.1.3. Libspe

La llibreria *libspe* és la llibreria que incorpora el Cell SDK (*Software Development Kit*, un paquet que incorpora un conjunt d'eines com ara compiladors, llibreries, etc. per treballar amb el Cell) que proporciona una interfície d'alt nivell per comunicar-se i utilitzar els processadors SPE.

En aquest apartat només detallarem el funcionament de tres funcions d'aquesta llibreria, que són les més utilitzades. Si es volen consultar més detalls, l'API està molt ben explicada al document oficial d'IBM, que es pot trobar a [30]. Per ara, les tres operacions que ens interessen són les següents:

- `spe_context_create`: Crea un context d'execució per l'SPE. Un context és una

abstracció que representa una tasca que s'executarà en un SPE. Seria com les `task_struct` que maneguen els processadors de propòsit general, que representen processos, però pels SPE, i disposarà tota la informació per començar o continuar una execució (estat dels registres, una còpia de la memòria, etc.). Per tant, es poden crear tants contextos com es vulgui, no només tants com SPE's hi hagi al sistema. Serà el planificador (*scheduler*) de tasques per l'SPE qui decidirà quina tasca s'ha d'executar. Aquesta operació fa la crida a sistema `spu_create`. Com nosaltres no definim un nou model d'execució, també necessitarem d'aquestes tasques específiques del nucli SPE.

- `spe_program_load`: A partir d'un context d'execució i un programa per l'SPE (encapsulat dins d'un binari CESOF), carrega el binari a la memòria de l'SPE corresponent.

- `spe_context_run`: Sol·licita l'execució del context i s'espera a que finalitzi. Per tant, és una crida que bloquejarà fins que finalitzi. Com aquesta operació la invocarà el PPE, serà aquest (el thread que l'executi) el que es quedi bloquejat fins que el SPE corresponent no notifiqui la finalització de l'execució. Aquesta operació fa la crida a sistema `spu_run`, que és la que realment és bloquejant.

Per tant, l'esquema típic d'una aplicació pel Cell (només la part que correspon al PPE) seria com el que es mostra, en pseudocodi, a la Figura 23. Com es pot apreciar, el programa principal (executat pel PPE) és l'encarregat d'anar invocant a la llibreria *libspe* tantes vegades com SPE's es vulguin utilitzar (suposant que aquest valor és un paràmetre modificable a l'hora d'arrencar el programa). L'usuari ha de ser qui creï el thread (en aquest cas, utilitzant la llibrera *pthread*) que executarà la funció `ppe_thread_func`, que invocarà la darrera crida explicada a la llibreria abans de morir.

En versions anteriors de la llibreria, la creació del thread la feia internament la pròpia llibreria, creant un *pthread* mentre el programa principal es quedava bloquejat. En posteriors versions s'ha refinat el codi, deslligant el paral·lelisme de la llibreria i permetent que l'usuari triï la opció que prefereixi.

```
main_PPE () {
     inicialitzar estructures de dades per guardar resultats execucions
     per cada SPE que es vulgui utilitzar {
          spe_context_create()
          spe_program_load()
          pthread_create (&ppe_thread_func,...)
     }
     envia dades als SPE
     espera finalització execucions
     rep resultats execucions
     exit
}

ppe_thread_func () {
     spe_context_run();
     pthread_exit();
}
```

Figura 23: Estructura típica d'un programa pel Cell

No es mostren en aquest exemple detalls sobre el procés de compilació, enviament de dades entre els nuclis, etc. tot i que no són del tot trivials. Cal utilitzar totes les eines que proporciona el Cell SDK per generar el binari pel PPE i l'SPE i acabar encapsulant-los en un binari CESOF. Només ens interessa veure com és el flux d'execució d'una aplicació per tenir una idea del model que es segueix actualment i com s'hauria d'adaptar perquè concordi amb la nostra proposta.

A la Figura 24 (extreta de [15]) es mostra un esquema d'execució al Cell. El thread principal crea quatre threads (contextos d'execució), que seran els encarregats de gestionar l'execució als coprocessadors SPE. Aquests threads d'usuari romandran bloquejats fins que no acabi l'execució als SPE's. El thread principal no es bloqueja, però haurà d'esperar que els threads que gestionen l'execució als SPE notifiquin la finalització de les execucions per poder recollir els resultats.

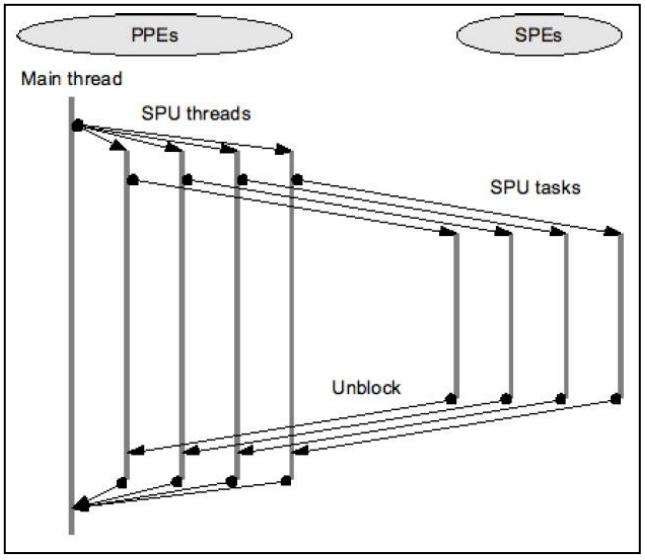


Figura 24: Execució de tasques als SPE's

7.2. Arquitectura PowerPC

PowerPC [29] és a una arquitectura RISC (*Reduced Instruction Set Computing*, que disposa de poques instruccions de codi màquina fent-lo estructuralment més simple) creada conjuntament per IBM, Apple i Motorola. Aquesta arquitectura és molt utilitzada actualment, tant en ordinadors personals (fins fa poc, abans de migrar cap a processadors Intel, els d'Apple incorporaven nuclis PowerPC) com en videoconsoles (les tres consoles de darrera generació, com són la Xbox 360 de Microsoft, la Wii de Nintendo i la PlayStation 3 que ens ocupa incorporen processadors PowerPC [31]).

Les principals diferències arquitectòniques respecte l'arquitectura x86 és que té molt poques instruccions de codi màquina (fent que el disseny sigui molt més senzill i eficient) i que els bancs de registres són més grans, cosa que contribueix al seu rendiment.

Una altra diferència, que ens podria portar problemes a l'hora de portar la nostra solució a una arquitectura PowerPC, és l'*endian*. L'*endian* és el format en què s'emmagatzemen les dades en un ordinador. Les dues alternatives existents són el *little-endian* i el *big-endian*. Aquestes alternatives serien com, per exemple en els llenguatges escrits, l'escriure de dreta a esquerra o d'esquerra a dreta. A la Figura 25 es presenta un exemple de cadascuna de les alternatives (imatges extretes de [32]).

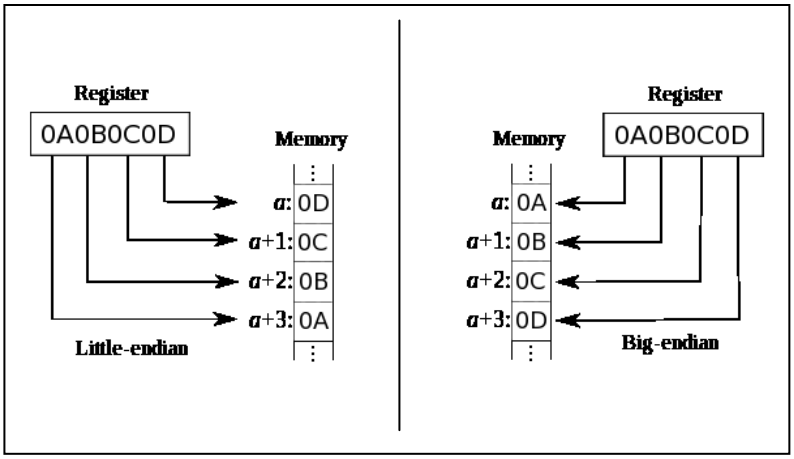


Figura 25: Diferència entre el format *little-endian* i el *big-endian*

Com es pot apreciar, el *byte* de més pes (*Most Significant Byte*, *MSB*), que en l'exemple correspon a `0x0A`, es guarda:

- A l'adreça de memòria més gran, en *little-endian*
- A l'adreça de memòria més petita, en *big-endian*

En canvi, el *byte* de menys pes (*Least Significant Byte*, *LSB*), que en l'exemple correspon a `0x0D`, es guarda:

- A l'adreça de memòria més petita, en *little-endian*
- A l'adreça de memòria més gran, en *big-endian*

Per tant, depenent de l'arquitectura sobre la qual s'estigui executant, les dades s'hauran de llegir i interpretar d'una manera o d'una altra. L'arquitectura x86 utilitza el format *little-endian*, mentre que l'arquitectura PowerPC utilitza *big-endian*. Per tant, aquest pot ser un problema en la migració (depenent de com s'hagi programat), i s'haurà de tenir en compte si això succeís.

Finalment, el nucli PowerPC present al Cell BE té una memòria cache de dades L1 de 32 KB i una d'instruccions L1 de 32 KB. La memòria cache de segon nivell és de 512 KB, i al banc de registres té disponibles 32 registres de 64 bits de propòsit general (*General-Purpose Registers*, *GPR*), 32 registres de 64 bits de coma flotant (*Floating-Point Registers*,

FPR) i 32 registres vectorials de 128 bits.

7.3. Proposta adaptada a una arquitectura PowerPC

En aquest apartat presentem les modificacions que s'han fet per adaptar la proposta per una arquitectura PowerPC. Com ja hem dit, en aquest apartat no tindrem en compte l'existència dels nuclis SPE's, ja que el simple canvi d'arquitectura ja ha fet que s'introdueixin algunes modificacions que és interessant explicar per separat.

Com s'ha fet al llarg de tota aquesta memòria, dividirem l'explicació en els tres grans grups implicats en el Projecte: nivell d'aplicació, llibreria HELF i sistema operatiu.

7.3.1. Nivell d'aplicació

En aquest apartat no hi ha hagut cap modificació rellevant. El format HELF no ha necessitat de cap modificació per suportar la nova arquitectura. L'únic que s'ha de tenir en compte és que, al ser un altre llenguatge màquina diferent al de l'arquitectura x86, cal compilar i enllaçar els programes de nou perquè el binari final sigui de tipus per PowerPC. A part d'aquesta peculiaritat, no ha fet falta tenir en compte res més per portar la solució a la nova arquitectura, ja que les crides a la llibreria HELF també es fan de la mateixa manera que s'ha explicat per una arquitectura homogènia x86. L'únic que caldrà és indicar que les funcions s'executaran en una arquitectura PowerPC.

7.3.2. Llibreria HELF

Quant a la llibreria HELF, l'única cosa que s'ha hagut de fer és generar versions idèntiques per 32 i 64 bits, donat que en aquesta etapa hem utilitzat binaris dels dos tipus i cadascun necessita la llibreria adequada. Les eines de compilació i enllaç actuals permeten generar fàcilment aquestes dues versions, i l'únic que cal fer és guardar cada versió al directori corresponent (`/usr/lib` en el cas de la llibreria de 32 bits, i `/usr/lib64` en el cas de la de 64 bits), on els binaris l'aniran a cercar.

7.3.3. Sistema operatiu

En aquest darrer apartat parlarem de les modificacions que s'han hagut de fer al sistema operatiu. En aquest cas sí que ha calgut fer diverses modificacions, que tot seguit s'expliquen. La idea és la mateixa que hem presentat fins ara, però s'han hagut d'adaptar diferents aspectes perquè tot funcionés correctament.

A continuació es presenten els problemes que han anat sorgint, i com s'han solucionat. La documentació que defineix l'ABI del format ELF per PowerPC de 64 bits ha sigut clau per resoldre gran part d'aquests problemes. Si el lector està interessat en més detalls sobre això, pot obtenir més informació consultant a [33]. Si es vol aprofundir en aspectes com ara assemblador PowerPC, també es pot consultar a [34].

Primer problema: Carregador HELF

El primer problema es va detectar al carregador del format HELF. A l'executar un binari HELF de 32 bits, el sistema retornava un error que no podia reservar una quantitat tant gran de memòria. En el cas d'un binari de 64 bits, la càrrega es feia correctament.

Aquesta memòria és la que utilitzem per guardar una secció del binari HELF, definida pel format ELF, on estan guardats els noms de totes les seccions. Com el lector pot intuir, aquesta secció ens serveix per anar mirant si les seccions comencen per `.text.`, detectant així que és una secció afegida pel format HELF. Sabent que aquesta mida no podia ser tant gran com perquè donés aquest error, vam buscar l'error, que havia de ser d'implementació.

El problema estava que no obteníem correctament el camp que guarda la mida d'una secció. Al sol·licitar una regió de memòria amb una mida no controlada, la càrrega era incorrecta. Aquest problema era a causa del fet que utilitzàvem una estructura genèrica que representa la capçalera d'una secció (a on hi ha la seva mida), independentment de si la secció pertanyia a un binari de 32 o 64 bits. Al compilar el kernel de la PlayStation 3 indicant que és una arquitectura de 64 bits, aquesta estructura genèrica quedava definida representant la capçalera d'una secció per binaris de 64 bits.

El compilador, per la seva banda, traduïa l'accés al camp que conté la mida mitjançant un desplaçament respecte l'adreça base de l'estructura (com fan els compiladors). Aquest desplaçament era correcte en el cas de binaris de 64 bits, però no en el cas de 32 bits, donat que els camps d'aquestes estructures no són de la mateixa mida i, per tant, el desplaçament per obtenir el camp que guarda la mida no és el mateix.

La solució que es va prendre va ser mirar de quin tipus era el binari (32 o 64 bits) i en funció d'això utilitzar una de les dues estructures concretes en temps d'execució, sense utilitzar l'estructura genèrica que es resol en funció de com es compila el kernel. Aquest error no succeeix al carregador estàndard de Linux per binaris ELF, perquè el carregador no treballa a nivell de secció, sinó de segment.

Per acabar de veure si la solució era correcta, es va analitzar el codi font de l'aplicació `readelf`, i vam comprovar que segueix la mateixa estratègia.

Va sorgir un altre problema, que s'explicarà tot seguit, que va fer que el carregador també es modifiqués. Per això donem la solució abans d'explicar el problema. En concret, a partir d'ara el camp que ens deia si el procés era de tipus HELF o no ja no serà un booleà, sinó que podrà prendre 3 valors:

- 0, en el cas que el procés no sigui HELF (com fins ara)
- 32, en el cas que el procés sigui HELF i representi un binari de 32 bits
- 64, en el cas que el procés sigui HELF i representi un binari de 64 bits

Com ja es pot veure, el fet que l'arquitectura sigui de 64 bits ha donat més problemes dels previstos.

Segon i tercer problema: Operacions específiques per transferir l'execució

Els dos següents problemes es van trobar en la funció específica per transferir l'execució a una arquitectura PowerPC. Recordem breument com funciona el flux d'execució fins que es fa el canvi de processador, per aclarir les idees i entendre millor els problemes trobats.

Es marcarà una funció per ser compilada per arquitectura PowerPC. Això quedarà reflectit al binari HELF, en una nova secció anomenada `.text.ppc`. El carregador HELF, ja arreglat, crearà el node que representa l'arquitectura PowerPC i farà que els seus camps `open` i `close` apuntin a les operacions específiques per transferir el control d'execució. Posteriorment, en temps d'execució, quan el programador vulgui que una determinada funció l'executi un nucli PowerPC, farà la crida a la llibreria HELF indicant-ho. La llibreria farà la crida a sistema `helf_execute`, que acabarà cridant a la funció específica per transferir el control a un nucli PowerPC.

El primer problema que explicarem es va trobar a la funció específica per transferir el control a un nucli PowerPC. En concret, funcionava correctament amb binaris de 32 bits, però donava un error de segmentació amb binaris de 64 bits (contràriament a l'error del carregador). Aquesta funció, com ja es va explicar per l'arquitectura x86:

1. Obté els registres de la pila de kernel del procés. En el cas de l'arquitectura PowerPC es guarden en un altre lloc, i ocupen molt més (hi ha més registres) però a part d'aquest fet no hi ha cap altra diferència.

2. Els guarda al nou camp de la `task_struct` (`HELF thread info`), juntament amb l'adreça on s'ha de guardar el resultat de la funció a executar. Aquest pas és comú en les dues arquitectures.

3. Canvia el registre `%nip`, (en PowerPC, l'equivalent a `%eip` en x86) que valgui el valor de la funció que volem executar.

4. Prepara els paràmetres que necessita la funció. En cas de x86 es guarden a la pila d'usuari, i en el cas de PowerPC es guarden en un registre de propòsit general (*GPR*).

Consultant informació trobada a [33] es va trobar la següent indicació: *"The value of a function pointer is the address of the function descriptor, not the address of the function entry point itself".*

És a dir, en el cas d'un binari de 32 bits, l'adreça d'una funció coincideix amb el punt d'entrada d'aquella funció, però això no es compleix en el cas dels binaris de 64 bits. Per tant, si canviem el registre `%nip` fent que valgui una adreça d'un binari de 32 bits ja funcionarà correctament, a diferència dels binaris de 64 bits, on aquesta adreça apunta a un descriptor, però en cap cas a la funció. És per això que els binaris de 32 bits funcionaven correctament, i no els de 64 bits.

Un descriptor de funció és una estructura que conté tres adreces, una de les quals és el punt d'entrada de la funció que descriu. Per tant, el pas 3 que s'acaba d'explicar varia lleugerament. En el cas d'un binari de 32 bits s'ha de fer que el registre `%nip` valgui l'adreça de la funció que volem executar, però en el cas d'un binari de 64 bits ha de valer el punt d'entrada de la funció obtingut mitjançant el descriptor de la funció.

Per aquest motiu, al carregador hem guardat si el procés representa un binari de 32 o 64 bits, per poder discernir en aquest punt en quina situació ens trobem i actuar conseqüentment.

El segon problema es va trobar a la crida a sistema `start_hexec`, invocada per la funció `helf_execute`. Aquesta crida a sistema, com ja es va explicar en l'arquitectura x86, realitzava aquests passos:

1. Comprovar si el procés que executa la crida és de tipus HELF. Si no és així, es retorna un error.

2. Obtenir el node que representa la ISA de l'arquitectura on es vol executar la funció (en el cas que ens ocupa, PowerPC). Si no existeix retornarem un error, donat que no podem dur a terme el salt sense la informació associada al node.

3. En cas d'existir el node, s'invoca el seu mètode `open`. Per tant, s'invoca la crida específica de l'arquitectura per saltar a una unitat d'execució PowerPC (la que acabem d'arreglar).

4. Al finalitzar el mètode específic, es retorna un 0 indicant que tot s'ha realitzat correctament.

El problema estava, precisament, en aquest quart pas. En l'arquitectura x86, el retorn d'una funció es guarda al registre `%eax`. En una arquitectura PowerPC, es guarda al registre `GPR[3]`. Això no comporta cap problema, aparentment. El problema és que l'arquitectura PowerPC defineix que els paràmetres de les funcions no es guarden a la pila d'usuari (com en el cas de l'arquitectura x86), sinó que es guarden als registres de propòsit general (*GPR*), començant pel registre `GPR[3]` i, si s'acabessin els registres, s'utilitzaria la pila d'usuari. Per tant, s'utilitza el mateix registre per guardar el retorn d'una funció que pel primer paràmetre d'una funció.

Com ja hem dit anteriorment, una de les feines que cal fer és preparar els paràmetres de la funció, perquè al sortir del sistema operatiu la funció se'ls trobés guardats a on els va a buscar. Ara bé, si abans de sortir del sistema operatiu retornem un 0, el que estem fent és sobreescriure aquesta informació al registre `GPR[3]`, fent que l'aplicació no disposi dels paràmetres. Un accés d'aquesta funció als paràmetres donarà un error de segmentació.

Per tant, en el cas de l'arquitectura PowerPC no s'ha de retornar un 0, sinó que s'ha de retornar precisament l'adreça dels paràmetres (que és un dels paràmetres de la crida a sistema, i per tant ja els tenim). Així, el compilador els guardarà automàticament al registre `GPR[3]`, sortirem del sistema operatiu retornant a usuari, i la funció es trobarà els paràmetres allà on els va a buscar.

Una vegada resolts aquests problemes, el comportament en una arquitectura PowerPC ha estat el mateix que la proposta per l'arquitectura x86, oferint a l'usuari les mateixes funcionalitats a nivell de llibreria i sistema operatiu.

7.4. Proposta final adaptada al Cell BE

En aquest quart apartat explicarem la migració completa del model a l'arquitectura Cell BE (utilitzant tant el PPE com els SPE). Aquí és on es fa realment el pas d'una arquitectura

multiprocessador homogènia a una d'heterogènia.

Aquesta darrera etapa no s'ha implementat tant a nivell de sistema operatiu com la resta. Fins ara, la llibreria era una capa intermèdia, que rebia les peticions de l'aplicació i les transmetia integrament al sistema operatiu, que era el que feia totes les funcions per canviar de processador. En canvi, en aquest cas d'integrar al sistema els coprocessadors SPE el pes important ha recaigut a la llibreria HELF. El motiu és que no hem tingut prou temps com per implementar la proposta a dins el sistema operatiu, tot i que això és factible. Però aquesta decisió no contradiu la filosofia seguida durant tot el Projecte, ja que la llibreria també és una part necessària en la nostra proposta. És per això que en aquest darrer apartat no hi ha hagut tants problemes d'implementació, ja que el que s'ha de fer és adaptar la llibreria HELF perquè dugui a terme els canvis d'execució als SPE.

En aquest apartat també dividirem l'explicació en tres apartats: nivell d'aplicació, llibreria HELF i sistema operatiu.

7.4.1. Nivell d'aplicació

El codi font de les aplicacions s'haurà de modificar per a que utilitzi la llibreria HELF. En aquest cas, el codi que correspon al PPE no haurà de fer les crides a la llibreria *libspe*, com l'exemple que s'ha presentat a la Figura 23, sinó que l'únic que haurà de fer és una crida a `helf_execute`, passant-li els paràmetres adequats. Com es comentarà a continuació, els paràmetres d'aquesta crida (funció a executar, paràmetres i adreça de retorn) no tenen la mateixa interpretació que tenien fins ara.

Seguint l'exemple presentat a la Figura 23, els nous programes per aprofitar la potència dels SPE's haurien de ser com el de la Figura 26. Com es pot veure, es creen un seguit de threads que canvien el seu context d'execució mitjançant la crida a la llibreria HELF. Si, pel contrari, volguéssim que fos el mateix thread principal qui canviés el seu context (el que s'ha anat proposant fins ara), l'únic que caldria fer és no crear aquests threads, sinó invocar directament a la crida a llibreria. D'aquesta manera, qui es quedarà bloquejat serà el thread principal, i no els threads que s'han creat.

```
main_PPE () {
     inicialitzar estructures de dades per guardar resultats execucions
     per cada SPE que es vulgui utilitzar {
          pthread_create (&ppe_thread_func,...)
     }
     envia dades als SPE
     espera finalització execucions
     rep resultats execucions
     exit
}

ppe_thread_func () {
     helf_execute (..., ARCH_SPE);
     pthread_exit();
}
```

Figura 26: Estructura típica d'un programa pel Cell adaptat per usar la llibreria

Aquestes modificacions, a més, necessitaran tot un seguit d'eines (similars a les que incorpora al Cell SDK, però modificant el seu comportament) que són les que generaran l'executable final tal i com estableix el format HELF. Aquestes eines, com ara *scripts* i Makefile's, hauran de ser utilitzats per l'usuari d'una manera força automàtica i còmoda, similarment a les eines que incorpora el Cell SDK. Per veure més detalls sobre aquests aspectes es pot consultar el Projecte Final de Carrera complementari a aquest, a l'apartat corresponent al paquet d'utilitats anomenat `HELFutils`.

7.4.2. Llibreria HELF

Com ja hem comentat, la gestió dels SPE's no es desenvolupa internament al sistema operatiu, sinó que serà la llibreria HELF qui farà aquesta feina. Com ja hem dit, això no contradiu la filosofia de la proposta, ja que els canvis de processador els duen a terme conjuntament la llibreria i el sistema operatiu. La llibreria HELF utilitzarà, a la seva vegada, la llibreria *libspe* per executar les funcions als SPE's.

Per tant, seguint la mateixa especificació que ja tenia la llibreria (que es pot consultar a la Taula 3), hem afegit un paràmetre més a la funció `helf_execute` per poder especificar tots els paràmetres que necessita la llibreria *libspe*, i hem canviat l'especificació de la resta de paràmetres (funció a executar, paràmetres de la funció i adreça de retorn) donant-li un nou sentit per adequar-los als paràmetres que necessiten les tres operacions de la llibreria *libspe*.

Podríem haver creat una nova crida de llibreria per no fer-ho tant complicat, però com ja es va comentar volíem que fos una única funció la que gestionés els salts entre processadors, per facilitar l'aprenentatge al programador.

D'aquesta manera, la llibreria HELF farà internament les tres crides explicades a l'apartat de la *libspe* per transferir el control d'execució, obtenint el resultat desitjat. Es crearà un context d'execució, que representarà la tasca a un SPE, se li enviarà el codi del programa i es transferirà el control d'execució fins que aquesta finalitzi. Per tant, es realitzen els passos habituals en un canvi de processador, utilitzant el model d'execució que ja està establert. Cal dir que, en aquest cas concret, no és necessari acabar l'execució a l'SPE amb una crida a `helf_exit`, ja que el canvi de processador el fa la *libspe* (recolzada per les crides a sistema que utilitza), i aquest retorn ja es fa automàticament. Tot i això, ja hem dit que en un cas genèric com el que es proposa sí que caldria fer-se.

No s'especifiquen en concret com canvien els paràmetres de la llibreria HELF, com tampoc s'han detallat els paràmetres de les tres funcions de la *libspe* que hi ha involucrades. Només hi ha un aspecte que cal tenir en compte, que s'explica tot seguit.

Com es va introduir a l'anàlisi d'antecedents d'aquesta memòria (apartat 3.1.1), el Cell utilitza el format executable CESOF, que el que fa és afegir dins d'una secció de dades l'executable per l'SPE. Aquesta no és, només, la única diferència respecte un executable ELF estàndard. En concret, el format CESOF crea uns punters que especifiquen on està situat aquest binari incrustat, de tal manera que el pugui localitzar quan l'hagi d'enviar per carregar-lo a la memòria local. Aquests punters estan inclosos en una estructura, anomenada *SPE Program Handle*. A la Figura 27 es presenta un exemple d'un binari CESOF amb aquesta estructura, extreta de [5].

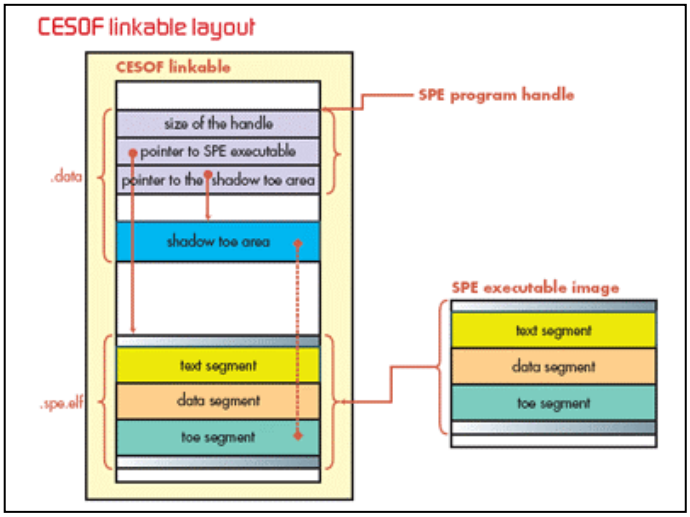


Figura 27: Exemple d'un binari CESOF

Es pot veure a la figura com hi ha l'adreça de l'executable SPE, que indica on està ubicat dins el fitxer CESOF.

Com el format HELF no té res a veure amb el format CESOF, aquestes dues diferències s'han d'eliminar. La primera ja es compleix, donat que el codi per l'SPE no es guarda en una secció de dades, sinó que es guarda (com defineix el format) en una secció de text, més concretament en la secció que porta per nom `.text.spe`.

La segona diferència s'hauria d'evitar no tenint aquests punters. Com la generació del binari CESOF és força complicada (els *scripts* que el generen no són gens intuïtius ni entenedors) ens podem conformar amb no utilitzar-los, com si no hi fossin. Per tant, com no els podem utilitzar però la llibreria HELF necessita conèixer on comença l'executable per l'SPE per comunicar aquesta adreça a la llibreria *libspe*, l'haurà de proporcionar el sistema operatiu mitjançant una nova crida a sistema, que és la única novetat a nivell de sistema operatiu que explicarem tot seguit.

7.4.3. Sistema operatiu

L'única novetat afegida per suportar l'execució en els SPE's és, com s'acaba de comentar, la crida a sistema que retornarà l'adreça on comença l'executable corresponent a l'SPE. La capçalera d'aquesta crida a sistema és la següent:

```
unsigned int get_spe_elf_image_addr ()
```

Com el format HELF defineix que els codis de les diferents ISA's estaran separats en seccions, i la informació d'aquestes seccions es guardaran en temps de càrrega del binari a les noves estructures de dades (anomenades nodes), l'únic que caldrà fer és identificar el node que representa l'arquitectura SPE i retornar l'adreça on comença, informació que tenim guardada al node.

Per tant, els passos que segueix aquesta crida a sistema són els següents:

1. Comprova si el procés que executa la crida és de tipus HELF. Si no és així, retorna un error.

2. Obté el node que representa l'arquitectura SPE. Si no existeix aquest node retornarem un error, donat que significa que el format HELF no té cap programa compilat per l'SPE. En cas d'existir el node, retornarem el camp `address`, que indicarà l'adreça on comença aquesta secció, que coincideix amb l'adreça on comença l'executable.

Amb aquesta crida a sistema s'ofereix la informació que la llibreria HELF necessita per dur a terme les execucions als acceleradors SPE.

7.5. Proves i resultats

En aquest apartat es presenten les proves que s'han realitzat en aquesta darrera etapa, juntament amb els resultats obtinguts. Com ja s'ha finalitzat la migració de la proposta, podríem dir que aquests són uns resultats definitius, que ens permetran avaluar realment el rendiment de tot el que s'ha implementat.

En primer lloc, comptarem el nombre de línies afegides al kernel de Linux, ara que la implementació del Projecte s'ha finalitzat, càlcul que ens ajudarà a conèixer la portabilitat de la nostra proposta.

En segon lloc, i tal i com vam fer amb l'arquitectura homogènia, avaluarem el *benchmark* Scimark2 al Cell, però sense tenir en compte la part dels SPE's (és a dir, el que correspondria al que s'ha explicat a l'apartat 7.3). D'aquesta manera s'avaluarà que la portabilitat a una nova arquitectura, com és PowerPC, s'ha realitzat correctament, avaluant la sobrecàrrega afegida. El *benchmark* només el provarem per la part de PowerPC perquè la seva adaptació al Cell és força complicada.

Finalment, avaluarem una implementació de l'algorisme de multiplicació de matrius feta pel Cell, utilitzant per aquest càlcul la potència que ofereixen els acceleradors SPE. Amb aquesta prova veurem si el sistema es comporta correctament en una arquitectura heterogènia.

7.5.1. Nombre de línies afegides

En aquesta primera prova volem comptar aproximadament quantes línies de codi font (*SLOC, Source Lines Of Code*) hem afegit al kernel de Linux. Això ens donarà una aproximació de si la nostra proposta és portable a altres entorns.

El nombre aproximat de línies afegides es detallen a continuació:

• Carregador HELF:	750 línies
• Funcions específiques de l'arquitectura:	100 línies
• Funcions auxiliars invocades de diferents punts del kernel:	50 línies
• Crides a sistema:	80 línies
• Definició d'estructures de dades i constants:	50 línies
TOTAL	1030 línies

Com es pot veure, gran part de les línies afegides corresponen al carregador HELF (gairebé un 75%), però ja vam comentar que és preferible mantenir camins separats entre la càrrega de binaris ELF i HELF per no interferir en el carregador estàndard de Linux.

Aproximadament, el kernel 2.6.25 de Linux (gairebé idèntic al que utilitzem) té 9,2

milions de línies de codi font [35]. Per tant, la nostra proposta suposa un increment d'un 0,0112%, un percentatge que considerem molt petit.

7.5.2. Scimark2 (PowerPC)

Com en el cas de les proves realitzades quan el Projecte estava implementat només per una arquitectura homogènia x86, l'aplicació que utilitzarem per provar que el sistema funciona correctament en un entorn PowerPC és el *benchmark* Scimark2. Es poden consultar tots els detalls d'aquest *benchmark* a l'apartat 6.4.2, on es va introduir més extensament al provar-lo a l'arquitectura x86. Les proves es realitzaran en una PlayStation 3.

Només recordarem que l'aplicació ja ens retorna una puntuació on aproxima el rendiment obtingut en l'execució (en *MFlops*, milions d'instruccions en coma flotant per segon), i juntament amb el temps d'execució seran els dos paràmetres que utilitzarem per comparar.

Com també es va fer en el cas de l'arquitectura x86, realitzarem dos tipus de proves: execució seqüencial i paral·lela (amb *pthreads*) dels cinc nuclis computacionals que componen el *benchmark*. Aquestes dues proves es faran amb la versió original del *benchmark* i amb la versió modificada utilitzant la proposta presentada per PowerPC. Com en totes les proves que s'han realitzat, l'entorn d'execució ha estat controlat, aïllant al màxim l'execució del *benchmark*, de tal manera que tots els recursos de la màquina estiguessin disponibles, i s'han fet prous proves com per veure que els resultats eren estables.

En aquesta primera prova, també succeeix el mateix que amb les proves fetes a l'arquitectura x86: no hi ha cap salt a cap processador, donat que estem ignorat els SPE's. Per tant, els resultats obtinguts entre les execucions originals i les que utilitzen el sistema proposat seran gairebé idèntics. Això ens ajudarà a concloure, com en el cas de l'arquitectura x86, que estem oferint una interfície còmoda per fer canvis entre processador sense penalitzar el rendiment de l'execució.

Rendiment d'una execució seqüencial, comparant les versions original i HELF

En aquesta primera prova comparem el rendiment dels cinc nuclis computacionals entre l'aplicació original i la versió HELF fent una execució seqüencial. A la Taula 7 es presenten els resultats d'aquesta prova.

Nucli computacional	Original	Versió HELF
FFT	28,86	29,07
SOR	108,11	108,21
MonteCarlo	9,28	9,33
Sparse matmult	34,47	34,92
LU	47,49	47,44

Taula 7: Comparativa de rendiment d'una execució seqüencial del *benchmark* Scimark2 (PowerPC)

El rendiment entre les dues versions és similar, ja que en un cas és més elevat el de la versió original (el nucli `LU`) mentre que en la resta ho és més la versió HELF. Però el rendiment obtingut és similar, cosa que indica que la nostra proposta no penalitza el rendiment de l'aplicació.

El que crida l'atenció en aquest cas és la diferència de rendiment obtingut entre les dues arquitectures. Com es podia apreciar a la Taula 5, quan vam presentar aquesta mateixa prova en l'arquitectura x86, veiem que els resultats obtinguts en l'arquitectura PowerPC són molt pitjors (de l'ordre de 5 vegades). Per tant, queda clar que el processador de propòsit general que incorpora la PlayStation 3 no obté un gran rendiment en l'execució de tasques, sinó que com ja vam dir, la seva funció hauria de ser distribuir la càrrega entre els SPE's.

Temps d'una execució seqüencial, comparant les versions original i HELF

A la Figura 28 es presenten els temps totals, en segons, de l'execució seqüencial comparant la versió original i la versió HELF.

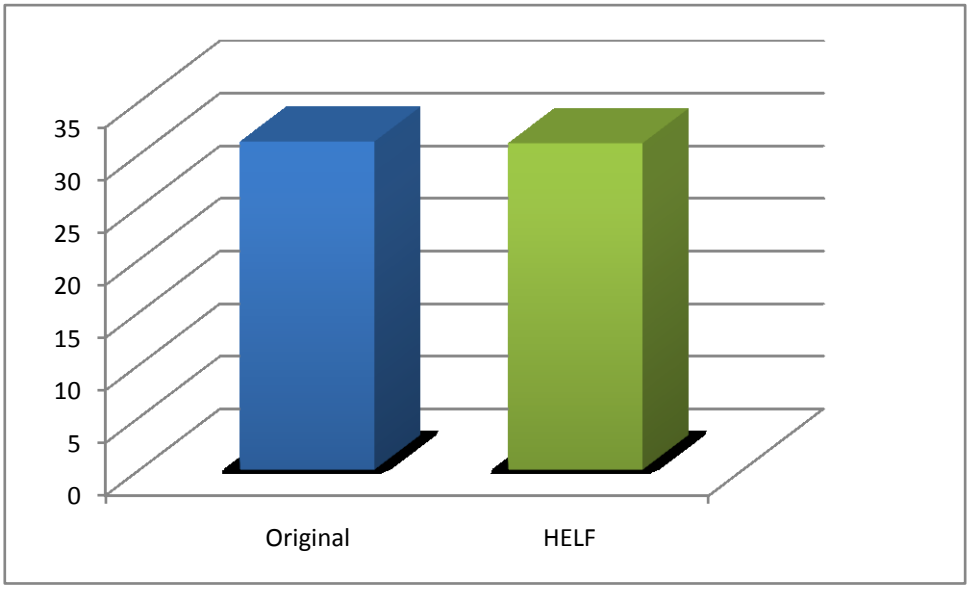


Figura 28: Comparativa de temps d'una execució seqüencial del *benchmark* Scimark2 (PowerPC)

En aquest cas, tarda lleugerament menys la versió utilitzant la nostra proposta que la versió original, tot i que la diferència és mínima. Comparant aquests temps amb els de l'arquitectura x86, veiem que són una mica més elevats.

Rendiment d'una execució paral·lela, comparant les versions original i HELF

En aquesta segona prova comparem el rendiment dels cinc nuclis computacionals entre l'aplicació original i la versió HELF fent que cada nucli l'executi un *pthread* paral·lelament. A la Taula 8 es presenten els resultats.

Nucli computacional	Original	Versió HELF
FFT	4,60	4,10
SOR	20,61	22,51
MonteCarlo	1,42	1,62
Sparse matmult	7,47	6,86
LU	7,62	7,83

Taula 8: Comparativa de rendiment d'una execució paral·lela del *benchmark* Scimark2 (PowerPC)

Com en el cas de l'execució seqüencial, el rendiment entre les dues versions és molt similar, i com també succeïa abans, les diferències entre aquesta execució i la mateixa en l'arquitectura x86 (veure Taula 6) són abismals. Per tant, ja veiem que el Cell no és una

bona plataforma per executar aplicacions que no treguin profit de la seva peculiar arquitectura heterogènia.

Temps d'una execució paral·lela, comparant les versions original i HELF

Finalment, a la Figura 29 es presenten els temps totals, en segons, de l'execució paral·lela, comparant la versió original i la versió HELF.

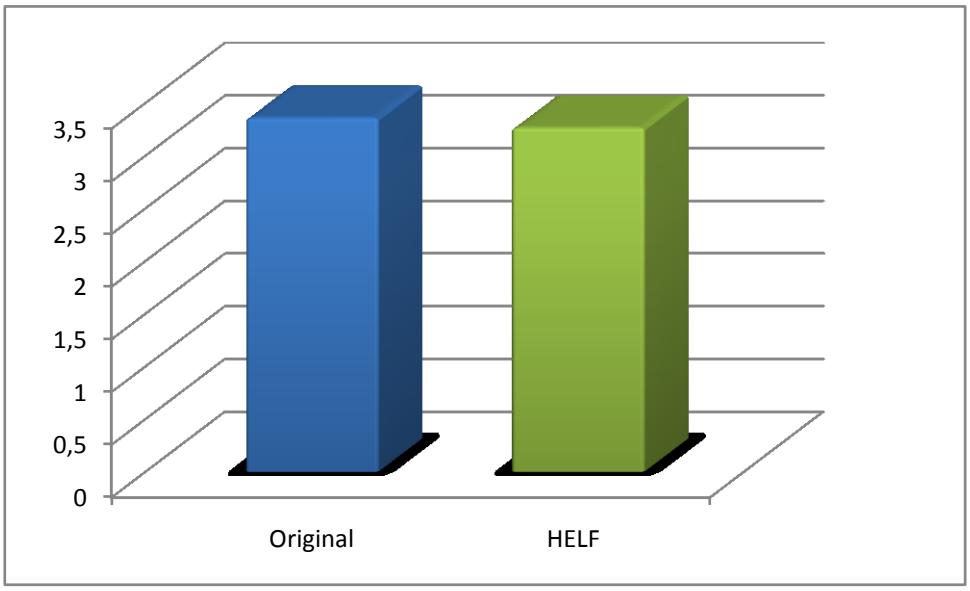


Figura 29: Comparativa de temps d'una execució paral·lela del *benchmark* Scimark2 (PowerPC)

En aquest cas succeeix el mateix que amb la prova feta en una arquitectura x86: el que crida més l'atenció es la diferència de temps respecte l'execució seqüencial. El fet de paral·lelitzar l'aplicació utilitzant *pthreads* fa que l'execució aconsegueixi un *speed-up* de 8X. Això deu ser a causa que el processador PowerPC és un nucli *SMT*, i pot executar dos fils alhora. Comparant entre sí la versió original i la versió adaptada, veiem que els temps d'execució són gairebé idèntics.

7.5.3. Multiplicació de matrius (Cell BE)

En aquest darrer apartat presentem les proves realitzades al Cell BE, és a dir, utilitzant els coprocessadors SPE. Com ja hem dit, la proposta d'aquesta darrera part no està integrada dins el sistema operatiu, sinó que s'ha integrat a la llibreria, que serà l'encarregada de transferir el control d'execució.

El *benchmark* que s'ha utilitzat per aquesta darrera part ha estat un ja disponible pel Cell, ja que adaptar una aplicació (com ara el *benchmark* Scimark2, que l'hem utilitzat per avaluar el sistema en una arquitectura homogènia x86 i PowerPC) per ser utilitzada al Cell és molt costós. Per tant, buscarem una aplicació que funcioni al Cell i l'adaptarem perquè utilitzi la nostra proposta.

El *benchmark* triat és una implementació d'una multiplicació de matrius, dut a terme per una universitat d'Alemanya, que es pot trobar a [36]. En resum, aquest *benchmark* calcula l'operació C = C + A * B; on A, B i C són matrius quadrades de mida múltiple de 128, i fa particions en blocs de 64x64 elements. Com a curiositat, dir que part de l'algorisme que executa el SPE està escrit en assemblador.

Aquest *benchmark*, com també succeïa amb l'Scimark2, ens retorna el rendiment (en aquest cas, és 1000 vegades superior, ja que la mesura es calcula en *GFlops*) de tots els SPE's que hagin intervingut en l'execució. A part d'aquest paràmetre, també mesurarem el temps d'execució. En aquest cas realitzarem dos tipus de proves: utilitzar un o quatre SPE's per calcular la multiplicació de dues matrius de 3200x3200 elements. Aquestes dues proves es faran amb la versió original del *benchmark* i amb la versió modificada utilitzant la proposta presentada. Com en totes les proves que s'han realitzat, l'entorn d'execució ha estat controlat, fent que es disposessin dels SPE's que es necessitaven perquè les proves no sortissin distorsionades.

Rendiment d'una execució, comparant les versions original i HELF amb un i quatre SPE's

En aquest cas no presentem cap taula, donat que els resultats amb les dues execucions han estat idèntics:

- El rendiment aconseguit utilitzant un SPE ha estat de 25,37 *GFlops*, tant la versió original com la versió adaptada HELF.
- Com amb un SPE, el rendiment del *benchmark* utilitzant quatre SPE's ha estat de 100,97 *GFlops* en la versió original i la versió HELF.

Per tant, ja podem veure que la nostra proposta no introdueix cap sobrecàrrega.

Tampoc apreciem en aquest cas una millora de rendiment, perquè estem comparant entre sí versions que utilitzem el mateix nombre d'acceleradors SPE. Però el que sí demostra és que podem enviar execucions als SPE's amb la nostra proposta, d'una manera còmoda (sense preocupar-se d'utilitzar la llibreria *libspe*), i s'aconsegueix el mateix rendiment que amb un programa programat específicament pel Cell.

Temps d'una execució, comparant les versions original i HELF amb un i quatre SPE's

A la Figura 30 es presenten els temps d'execució, en segons, del *benchmark* utilitzant un i quatre SPE's. Es comparen entre sí els resultats obtinguts amb la versió original i la versió HELF.

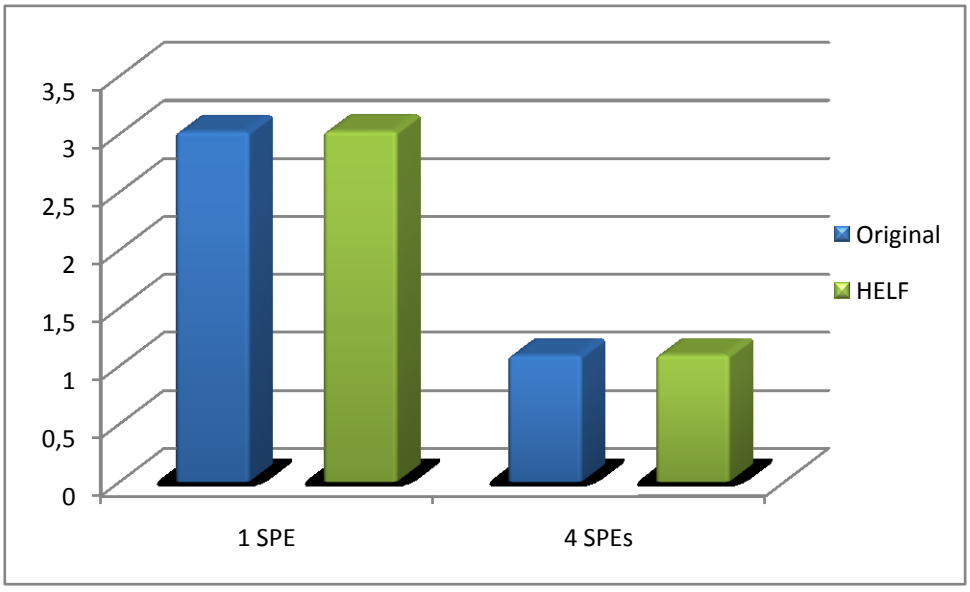


Figura 30: Comparativa de temps de les execucions utilitzant un i quatre SPE's

Com es pot apreciar, les diferències entre utilitzar la versió original i la versió adaptada són pràcticament inexistents, mentre que s'aprecia molt el fet de dividir o no el càlcul de la multiplicació de la matriu entre els diferents coprocessadors, ja que la diferència de temps és d'una tercera part aproximadament (uns tres segons amb un sol SPE i un segon utilitzant quatre SPE's). Per tant, és evident que el dividir la feina entre els coprocessadors fa que el càlcul es realitzi més ràpidament, tot i que la reducció no és lineal (amb quatre SPE's el càlcul es realitza tres vegades més ràpid).

7.6. Conclusions

En aquest darrer apartat presentem les conclusions a les que s'ha arribat amb tot el presentat en aquest capítol, on s'ha presentat la migració de la proposta a una arquitectura heterogènia com la Cell BEA.

Els dos primers apartats d'aquest capítol han estat per introduir l'entorn de treball, i després s'ha implementat la migració. Aquesta migració s'ha dut a terme en dues fases: primer s'ha adaptat tota la infraestructura per poder ser utilitzada en una arquitectura PowerPC, sense tenir en compte els coprocessadors que fan que el Cell sigui un multiprocessador heterogeni, i en segon lloc s'ha acabat de refinar la migració permetent l'execució de codi en aquests coprocessadors.

De la primera part, hem pogut apreciar les dificultats trobades a nivell de sistema operatiu pel simple fet de canviar d'arquitectura. Per exemple, el fet de ser una arquitectura de 64 bits ha comportat més problemes del que es preveia, mentre que la diferència d'*endian* entre x86 i PowerPC no ha implicat cap complicació especial. A la resta de parts que integren el Projecte (aplicació i llibreria) no s'han trobat complicacions en aquesta primera part donat que no s'ha requerit cap modificació.

En la segona part, en canvi, qui ha necessitat més modificacions ha estat la llibreria. S'han redefinit els paràmetres de les crides a la llibreria, fent que equivalguin als paràmetres que es necessiten per dur a terme una execució als SPE's, definits per la llibreria *libspe*. Per tant, la proposta no s'ha introduït al sistema operatiu, com tot el que s'havia realitzat fins ara, sinó que per comoditat s'ha implementat a nivell de llibreria. Seria més adient amb la nostra proposta que es fes al sistema operatiu, si es volgués continuar el desenvolupament en un futur. Per fer això, podríem utilitzar i partir de la proposta presentada a [15] perquè el sistema operatiu tingués una visió homogènia dels SPE, i utilitzant tot el que ja s'ha fet en aquest Projecte, fer la gestió i els canvis d'execució internament tal i com s'ha fet amb les arquitectures homogènies.

El codi font dels programes, per tant, s'han hagut d'adaptar, traient tota la interacció amb la llibreria *libspe* i fent que els canvis de processador els realitzi la llibreria

implementada per nosaltres.

Finalment, s'han provat aquestes dues parts separadament, i hem pogut verificar el que ja s'havia vist en les etapes anteriors: la nostra proposta proporciona flexibilitat al programador per indicar un canvi de processador en el flux d'execució d'una aplicació sense afegir una penalització excessiva. Per tant, hem complert amb el principal objectiu d'aquest Projecte Final de Carrera.

8. CONCLUSIONS

En aquest capítol es presenten les conclusions a les que s'ha arribat després de realitzar tot el projecte. En primer lloc, comentarem les tasques que s'han realitzat, que vindria a ser un resum de la proposta que es va fer a l'inici del Projecte, i s'aprecia com s'han acomplert els objectius que es perseguien. En segon lloc, explicarem l'experiència obtinguda al llarg d'aquests mesos amb tot el treball desenvolupat. Finalment, descriurem els passos que caldria seguir i els aspectes que caldria tenir en compte si es volgués afegir una nova arquitectura al sistema.

8.1. Tasques realitzades

Durant la realització del Projecte Final de Carrera s'han estudiat les arquitectures multiprocessador heterogènies, realitzant una proposa que, partint des del nivell d'aplicació i acabant pel sistema operatiu, permet transferir el control d'execució d'un processador a un altre segons les necessitats que indiqui el programador. Al ser un Projecte Final de Carrera basat en la recerca, molta part s'ha invertit en estudiar propostes existents, analitzar les tendències actuals del mercat i proposar diferents alternatives per avaluar els seus avantatges i inconvenients, realitzant una proposta convincent que afecta transversalment a diverses àrees de l'arquitectura de computadors. Part d'aquesta proposta s'ha acabat implementat per poder ésser provada.

Basant-nos en propostes i idees existents, com el model de continuacions del microkernel Mach i dels manegadors de dispositius de Linux, hem proposat un nou mètode per gestionar i executar binaris amb codi heterogeni. Començant pel nivell d'aplicació, hem modificat el codi font de les aplicacions per adaptar-lo als requeriments ja presentats que té el nostre sistema (pas de paràmetres, retorn, etc.) i hem proporcionat al programador una interfície perquè pugui indicar característiques d'un tros de codi, a nivell de funció.

Per altra banda, hem implementat una llibreria d'usuari per facilitar al programador la gestió del flux d'execució d'un thread. Aquesta llibreria només ofereix suport per entorns heterogenis, però és totalment compatible amb altres llibreries que podrien oferir, per exemple, paral·lelisme a l'aplicació. Per tant, no ens volem lligar a utilitzar alguna implementació per manegar fils d'execució, i la decisió la prendrà l'usuari en funció de les

seves preferències.

Finalment, a nivell de sistema operatiu hem afegit la capacitat de reconèixer aquests nous binaris, per extreure la nova informació afegida pel programador, guardar-la en unes estructures de dades associades al procés i utilitzar aquesta informació en temps d'execució per dur a terme els canvis d'unitat d'execució sol·licitats per la llibreria, mitjançant les funcions específiques de cada arquitectura per canviar el context d'execució.

Moltes de les decisions de disseny que s'han pres s'han basat en propostes ja existents a Linux (esquema de les crides a sistema amb un punt d'entrada comú, manegadors de dispositius, continuacions...), fent que la proposta sigui robusta i portable.

Hem provat i analitzat aquest projecte tant a nivell de sistema operatiu com a nivell d'usuari, i els resultats són molt optimistes. La proposta no afegeix gairebé penalització ni al sistema operatiu ni a nivell d'aplicació. La majoria de tests s'han fet en processadors de propòsit general, ja que el volum de treball que ha implicat tot l'estudi necessari ha sigut prou gran com perquè no donés temps en un període reduït com és la realització d'un Projecte Final de Carrera. Per tant, el que s'ha fet és una petita aproximació al que s'ha proposat, i a partir d'aquest punt es podria continuar el desenvolupament amb les idees que es proposen al capítol següent com a treball futur.

Aquests tres grans blocs són els que ja ens vam plantejar a l'inici del Projecte, i tot i que han anat variant certs aspectes a mesura que hem anat aprofundint més en l'estudi i rebent *feedback* dels revisors dels congressos a on hem enviat els nostres articles, la idea principal s'ha mantingut intacta i podem dir que ha acabat fructificant.

D'aquest manera, també en el Pla de Formació s'han assolit els objectius esperats, ja que s'ha seguit la metodologia pròpia de la recerca, s'ha relacionat i utilitzat informació de moltes assignatures de la carrera, s'ha participat en congressos i jornades de discussió i s'han redactat i publicat diversos articles en congressos que es realitzen arreu del món, fets que posen de manifest que la idea proposada és adequada i que el treball realitzat és satisfactori.

8.2. Experiència obtinguda

Quant a l'experiència personal obtinguda, aquest Projecte m'ha servit per aprofundir en temes d'arquitectura de computadors i sistemes operatius, que espero que em serveixin en un futur professional a nivell empresarial. En concret, aprendre més aspectes de l'arquitectura x86 i iniciar-me en noves arquitectures com són PowerPC i Cell BEA (de les quals no tenia cap coneixement fins al moment) m'ha servit per començar a conèixer una mica més aquestes arquitectures, que sembla que tenen un futur interessant.

També estic molt orgullós per haver publicat diversos articles de caire tècnic durant la meva època universitària, i més encara que s'hagin acceptat en congressos on assisteixen grans personalitats del món de l'arquitectura de computadors. Crec que és molt important haver aprofitat aquesta oportunitat que, gràcies al Pla de Formació, he tingut. Tot i que no continuaré la meva carrera professional dedicant-me a la recerca, he viscut aquesta etapa de la meva vida amb molta intensitat i crec que ha estat molt profitosa per mi, i és una bona manera d'acabar la carrera.

8.3. Passos a seguir per afegir una nova arquitectura

En aquest darrer apartat explicarem quins aspectes cal tenir presents si volguéssim afegir una nova arquitectura al sistema. Per fer-ho, presentem els passos que caldria seguir per fer-ho.

En primer lloc, quan sapiguem quina arquitectura volem afegir, cal definir el seu nom i codi numèric perquè les tres parts implicades (aplicació, llibreria i sistema operatiu) la coneguin i es puguin entendre entre elles, com es va presentar a la Taula 2.

Una vegada fet això, el binari ha d'incorporar una nova secció de text, el nom de la qual seguirà el format establert `.text.nomISA`, que incorporarà el codi compilat amb aquella ISA de totes les funcions que s'hagin marcat per ser executades en aquella arquitectura.

Després, caldrà modificar el carregador HELF perquè reconegui aquesta nova secció,

i creï un node que la representi. La informació que ha de guardar en aquest node és la que ja s'ha presentat, i els camps més importants són els punters a les operacions per canviar el context d'execució, que hauran d'apuntar a dues noves funcions. A part d'això, s'hauran de tractar aspectes com el planificador de processos, la migració de tasques entre els diferents nuclis d'execució, etc.

Aquestes dues funcions seran específiques de la nova arquitectura que estiguem afegint. No es poden donar uns passos en concret per fer això, donat que cada arquitectura té les seves peculiaritats. Caldrà, doncs, veure com és l'arquitectura en aspectes com:

- Registres del processador
- 32 o 64 bits, *little-endian* o *big-endian*
- Pila d'usuari i kernel
- Memòria compartida o local; carregar el codi i/o les dades
- On es guarden els paràmetres i retorn de les funcions
- Representació de les tasques a memòria per aquell processador; si disposa d'un planificador específic

Com ja es veu, no es poden generalitzar les tasques que han de fer aquestes operacions, ja que dependran de l'arquitectura. Per tant, caldrà un aprenentatge previ profund per conèixer la nova arquitectura, per poder dur a terme aquests canvis amb garanties.

Un cop això estigui fet, el programador només haurà d'indicar que vol executar una funció a la nova arquitectura mitjançant el seu codi numèric (o la constant equivalent que s'ha definit per facilitar la interacció de l'usuari amb la llibreria), i la llibreria no caldrà modificar-la, ja que farà la crida a sistema que cercarà el node que representa aquesta arquitectura i invocarà la funció específica per canviar el context d'execució. Per aquest motiu volíem tenir agrupats tots els canvis de context d'execució sota la mateixa crida a llibreria, per facilitar l'aprenentatge a l'usuari i millorar l'escalabilitat del sistema.

9. TREBALL FUTUR

Els objectius plantejats a l'inici del Projecte s'han complert satisfactòriament, tot i que hi ha una sèrie de punts que, per falta de temps, no s'han afrontat amb la profunditat desitjada. A més, a mesura que s'ha anat desenvolupant el projecte han sorgit idees que podrien servir si es volgués continuar aquest treball. Aquests dos aspectes són els que es presenten en aquest darrer capítol d'aquesta memòria.

9.1. Coses a millorar

Com tot Projecte, hi ha punts més dèbils que altres, però l'important és detectar quins són aquests punts i què s'hauria de fer per posar-hi remei. Per aquest motiu, hem analitzat la nostra proposta i hem detectat els dos punts més fluixos, que creiem que serien els primers que s'haurien de treballar i revisar si es decidís continuar amb aquesta proposta.

9.1.1. Generació del binari

En primer lloc, la generació del binari és força manual, tot i que seria interessant implementar un compilador que interpretés les marques afegides, compilés cada funció per la ISA determinada, agrupés aquests codis binaris en seccions de text tal i com defineix el format HELF i tot s'empaquetés dins d'un binari HELF, amb el seu nombre màgic. Tot això, com es pot intuir, comportaria una feina a nivell de compilador i enllaçador que no eren objectius d'aquest Projecte, ni es disposava de temps per fer-ho, però serien aspectes interessants per treballar-hi. Faria falta, gairebé, un tercer Projecte Final de Carrera que s'ocupés d'aquest aspecte.

9.1.2. Estudi d'arquitectures heterogènies

En segon lloc, s'ha aprofundit poc en una arquitectura heterogènia com el Cell BEA. La feina que ha portat pensar i dissenyar la proposta, implementar-la en una arquitectura homogènia x86 i migrar-la a una arquitectura homogènia PowerPC i el reduït temps que representa un Projecte Final de Carrera no han permès aprofundir en totes les possibilitats que ofereix el Cell BE. Hem utilitzat els mètodes que ofereix la llibreria *libspe*, però el que seria interessant és portar tota la gestió dels coprocessadors SPE's a dins el sistema operatiu.

9.2. Idees que es podrien desenvolupar

Una vegada explicats els punts febles que hem trobat volem presentar idees que, a mesura que s'ha aprofundit en el Projecte, han anat sorgint i que seria interessant plantejar-se si es vol continuar amb la realització d'aquesta proposta, dotant-la de més funcionalitats.

9.2.1. Versions de la mateixa funció

La primera idea que es proposa és la de disposar de diferents versions per la mateixa funció. Com ja vam comentar, el programador podria marcar una funció indicant més d'una arquitectura, i aquesta funció es podria compilar per diferents ISA's. Al carregar el binari, podríem construir una estructura com la que es presenta a la Figura 31.

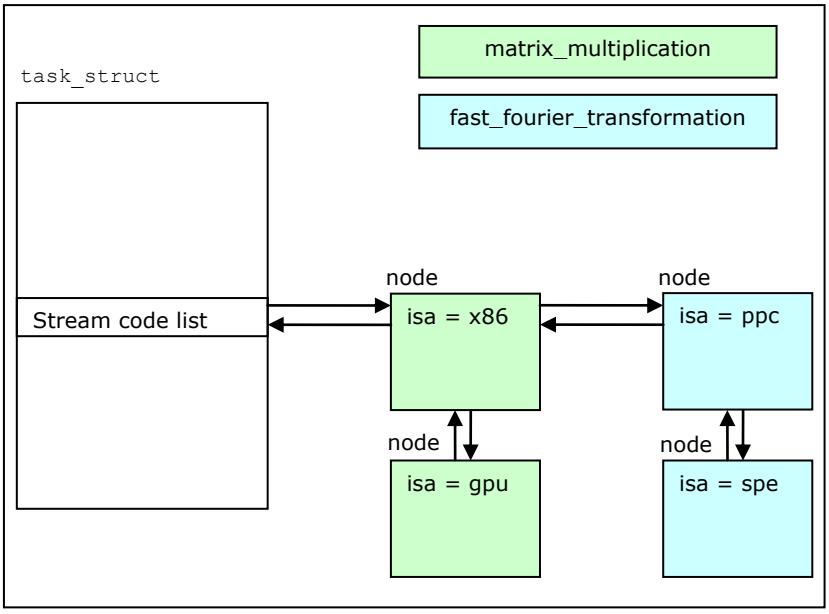


Figura 31: Ampliació del sistema amb suport pel manteniment de versions per funció

Com es pot veure, ara cada node no representa una ISA, sinó que representa una funció (com en la primera aproximació que vam fer), i existeixen dos tipus d'enllaç: un a les diferents versions de la mateixa funció i l'altre el que ja existia fins ara. D'aquesta manera, tenim agrupades totes les versions generades per les diferents arquitectures. En l'exemple, tenim dues funcions anomenades `matrix_multiplication` i `fast_fourier_transformation`, que s'han compilat per dues ISA's cadascuna.

En temps d'execució, el programador podria fer dues coses mitjançant la llibreria ja

existent:

- Indicar al sistema que es vol executar una funció en una arquitectura en concret (com fins ara).
- Indicar al sistema que es vol executar una funció, però sense especificar l'arquitectura sobre la que ha d'executar-se. En aquest cas, el sistema coneix quines possibilitats té (mitjançant les versions) i podria decidir en temps d'execució on és més adient que s'enviï aquest càlcul (depenent de l'estat dels nuclis o del rendiment que cada nucli traurà de l'execució, per exemple).

9.2.2. Comprovar les arquitectures en temps d'execució

Com ja es va comentar quan es van introduir les funcions de la llibreria HELF (veure Taula 3), hi ha una operació anomenada `valid_arch` que diu si una arquitectura és vàlida. Aquesta operació, ara per ara, només comprova si el codi que té com a paràmetre és un codi definit, independentment de si la màquina disposa o no d'una unitat d'execució d'aquell tipus. La possible ampliació seria comprovar en temps d'execució si l'arquitectura és vàlida per la màquina on s'executa, és a dir, que disposi d'una unitat d'execució d'aquell tipus.

Per fer-ho així, caldria suport per part del sistema operatiu, i un coneixement extens de les estructures de dades que manega Linux per gestionar els processadors internament. Un Projecte per homogeneïtzar la visió dels processadors per part del sistema operatiu, com el presentat a [15], ens seria de gran utilitat en aquest cas.

Fent-ho així, l'operació que comprova l'arquitectura seria més útil per comprovar si el canvi de context d'execució realment es pot dur a terme.

9.2.3. Gestió dels SPE's i precàrrega de l'executable

Finalment, en el cas d'una arquitectura com el Cell ja hem comentat que s'haurien de gestionar internament els coprocessadors SPE. Caldria, doncs, disposar d'alguna estructura de dades on ens poguéssim anotar l'ocupació d'aquests processadors i gestionar

internament els contexts d'execució, prescindint totalment de la llibreria *libspe*. Per fer-ho caldria conèixer exactament totes les tasques que aquesta fa i fer els passos equivalents, però a dins el sistema operatiu. En aquest cas també ens seria de gran utilitat el Projecte Final de Carrera que estava enfocat a homogeneïtzar la visió que té el sistema operatiu de tots els coprocessadors del Cell BE [15].

Una altra ampliació que es podria dur a terme, si aquesta primera part estigués feta, és detectar en temps de càrrega si un binari HELF disposa d'un programa incrustat per un SPE (si trobem la secció `.text.spe`), i aprofitar la càrrega del binari HELF per precarregar en la memòria local d'un SPE el seu codi i recordar a quin SPE s'ha enviat el codi. D'aquesta manera, quan es volgués llançar l'execució ja tindríem carregat el binari a l'SPE.

10. REFERÈNCIES I BIBLIOGRAFIA

[1] **Intel® Core™2 Duo**: http://www.intel.com/products/processor/core2duo/index.htm

[2] **Intel® Core™2 Quad**: http://www.intel.com/products/processor/core2quad/index.htm

[3] P. Hester. ***Multicore and Beyond: Evolving the x86 Architecture***. Symposium on High Performance Chips. 2007.

[4] **Fedora Core**: http://fedoraproject.org/

[5] A. Chow. ***Programming the Cell Broadband Engine***. Embedded Systems Design magazine. 2006.

[6] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G. Lueh, H. Wang. ***EXOCHI: Architecture and Programming Environment for A Heterogeneous Multi-core Multithreaded System***. Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. 2007.

[7] P. Bellens, J. M. Perez, R. M. Badia, J. Labarta. ***CellSs: a Programming Model for the Cell BE Architecture***. Proceedings of the 2006 ACM/IEEE conference on Supercomputing. 2006.

[8] L. Dagum, R. Menon. ***OpenMP: An Industry-Standard API for Shared-Memory Programming***. 1997.

[9] J. J. Dongarra, S. W. Otto, M. Snir, D. Walker. ***A message passing standard for MPP and workstations.*** 1996.

[10] N. Zea, J. Sartori, R. Kumar. ***Servo: A Programming Model for Many-core Computing***. 2007.

[11] **The Cell Project**: http://www.research.ibm.com/cell/

[12] **IBM Systems Journal**: http://www.research.ibm.com/journal/sj/

[13] **Fat Binary**: http://developer.apple.com/documentation/DeveloperTools/Conceptual/

MachORuntime/Reference/reference.html

[14] TMurgent Technologies. ***Processor Affinity - Multiple CPU Scheduling***. 2003.

[15] J. M. Merino. ***Eficient Resource Management in Heterogeneous Multiprocessor Systems***. UPC-PFC. 2007.

[16] M. Gil, L. Alvarez, X. Joglar, J. Planas, X. Martorell. ***Operating System Support for Heterogeneous Multicore Architectures***. UPC-DAC-RR-CAP-2007-40. 2007.

[17] X. Joglar, J. Planas, M. Gil. ***Adapting ELF to Load Heterogeneous Binaries***. UPC-DAC-RR-2008-6. 2008.

[18] **Third Workshop on Software Tools for MultiCore Systems (STMCS 2008)**: http://web.mit.edu/rabbah/www/conferences/08/stmcs/

[19] X. Joglar, J. Planas, M. Gil. ***OS Paradigms Adaptation to Fit New Architectures***. Proceedings of the Fourth Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA). 2008.

[20] **Fourth Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA 2008):** http://www.ideal.ece.ufl.edu/wiosca/

[21] **Annual Technical Conference USENIX 2008**: http://www.usenix.org/events/usenix08/index.html

[22] Tool Interface Standards (TIS) Committee. ***Executable and Linking Format (ELF) Specification, version 1.2***. 1995.

[23] J. Levine. ***Linkers and Loaders***. Elsevier Science & Technology Books. 1999.

[24] T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. M. Levy. ***Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism***. ACM Transactions on Computer Systems, Vol. 10, No. 1. 1992.

[25] D. Povet, M. Cesati. ***Understanding the Linux Kernel, 3rd Edition***. O'Reilly Media. 2005.

[26] IBM Cell BEA. **Linux Reference Implementation ABI Specification, version 1.2**. 2007.

[27] R. P. Draves, B. N. Bershad, R. F. Rashid, R. W. Dean. ***Using Continuations to Implement Thread Management and Communication in Operating Systems.*** Technical

Report CMU-CS-91-115, Carnegie Mellon University. També apareix a Proceedings of the Thirteenth Symposium on Operating Systems (SOSP). 1991.

[28] J. Corbet, A. Rubini, G. Kroah-Hartman. ***Linux Device Drivers, 2nd Edition***. O'Reilly Media. 2001.

[29] **IBM PowerPC**: http://www-03.ibm.com/technology/ges/semiconductor/power

[30] IBM Cell BEA. **SPE Runtime Management Library, version 2.2**. 2007.

[31] X. Joglar, J. Planas. ***Consoles de nova generació***. Treball Arquitectura d'un PC, curs 2006/2007 Q1.

[32] **Wikipedia, Endianness:** http://en.wikipedia.org/wiki/Endianness

[33] I. L. Taylor. ***64-bit PowerPC ELF Application Binary Interface Supplement 1.9***. 2004.

[34] **Assemblador PowerPC**: http://www.ibm.com/developerworks/linux/library/l-ppc/

[35] **Wikipedia, Linux kernel**: http://en.wikipedia.org/wiki/Linux_kernel

[36] **Fast Matrix Multiplication on Cell**: http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/matmul/